Bachelor Thesis

Final Report

Expert System for drug dosage adaptation

Author:
Nadir Benallal

Supervisor:
Yann Thoma

Expert:
François Veuve

26th October 2018

Abstract

This document presents the work done for the Bachelor thesis: Expert System for drug dosage adaptation. The subject of this project consists in adding a server layer on an already existing software called *Tucuxi* that will serve as an expert system for therapeutic drug monitoring (TDM).

TDM is a part of the work of a pharmacologist. It consists in monitoring the drug concentration in the blood of the patients. Some drugs have a small or particular target range and being outside of this range can be inefficient or dangerous for the patient. So the pharmacologist analyzes blood samples to be sure of the good health conditions of the patient.
This is a time consuming and difficult activity. A lot of parameters have to be taken into account. It also means that the patient can wait a while for the results of his blood sampling and that the explanations can be difficult to understand for him.

The *Tucuxi* software analyzes drug concentrations in blood samples and can compute drug concentration predictions, suggest dosage adjustments and evaluate a drug concentration. Thanks to these features it would be possible to add a layer to the software that could do some TDM. This would allow health practitioners to do TDM themselves, thus gaining a lot of time because they do not need to ask to a pharmacologist anymore. However the server must be thought so that it can respond to the requests quickly.

An analysis of the server needs, features provided and architecture have been done. This analysis mainly consists in discussions about the different options available for the server architecture taking into account that the requests received should be answered quickly.
Then some design choices were made about the features provided by the server and communication with it. The features provided by the server are based on the typical requests sent to pharmacologists for TDM. Some design choices had also to take into account that the server's architecture could change and it should not be difficult to change it.
When these design choices were made, an attempt at creating the server was made. The foundation of most of the designed features have been set up, but the server is not yet able to send a response with the expected output from the request received.

Contents

Chapter 1

Introduction

This document presents the work done for the Bachelor thesis: Expert System for drug dosage adaptation. The subject of this project consists in adding a server layer on an already existing software called *Tucuxi*. This software aims for blood sample analysis. It offers features such as drug concentration evaluation, prediction and drug dosage adaptation.

The server layer is an expert system that responds to some typical questions asked to pharmacologists in their therapeutic drug monitoring (TDM) work. TDM is a part of the pharmacologist work that aims at monitoring the drug concentration in the blood of the patients for drugs with a small or particular safe value range. The pharmacologist compares the blood drug concentration to the population standard expected values or more specific range values depending on the health condition of the patient like his age, weight, an organ malfunction or his illness history. He can then suggest better dosage alternatives or confirm the actual dosage depending on the the blood sample value being inside or outside the consulted safe range values.
This work is useful for drugs with small safe range values or for drugs that are not used often. Some of these drugs can be dangerous for the patient if the drug concentration is under or over the safe range or it can also become ineffective. Doctors that use this type of medical treatment sometimes prefer or must ask a pharmacologist for a TDM report.

TDM is difficult because a lot of parameters must be taken into account when analyzing the drug concentration in a blood sample like, for example: the date at which the drug was given, the route of administration, the dosage, when the sample was taken, when the sample was analyzed in the laboratory, etc. It is a time consuming work and the doctors can't do this type of work in addition to what they already have to do.
Thanks to a software that analyzes the treatment of a patient automatically and quickly, both the pharmacologists and other health practitioners can gain a lot of time to spend it more with the patient or in the rest of their work.

The goal of this project is to create an expert system server that answers the same questions that a pharmacologist typically does in a TDM session. Thanks to the *Tucuxi* software on which the server layer will be added, the server must be able to receive the patient's medical data and process them to predict the future drug concentration in the patient's blood, judge the risk of his actual dosage and suggest a first dosage of a dosage adaptation based on these predictions.
With this feature added to the *Tucuxi* software, the health practitioners will be able to do the drug monitoring themselves after the blood sample they have sent to the hospital has been analyzed and added in the hospital database. This makes the feedback to the patient faster and clearer thanks to the *Tucuxi* GUI. The health practitioner can also take more time to explain clearly the results of the TDM with the help of the report generated by the *Tucuxi* software.

In the second chapter of this document we present the analysis of the project consisting of the context of the project, its goal, a brief list of the server's functionalities, a discussion on the architecture choices, a brief description of the technologies involved and some remarks on the TDM work observed at *CHUV*.

In the third chapter we expose some design choices on the server's functionalities, the REST API endpoints and the communication between the client and the server.

In the fourth chapter we present how the server has been implemented: the code organisation, the server configuration, how the REST API is implemented as well as the serialization/deserialization and the computing of the requests sent to the server.

In the fifth chapter we show how the test have been done and their procedures.

In the sixth chapter we present the initial planning of the project, followed by the real proceedings of the project until the return of this document. A brief conclusion of the document is presented at the end of the chapter.

Chapter 2

Analysis

An analysis of the server needs, features and architecture must be done and discussed in order for the server to be as complete as possible. It is also important to know in advance what kind of problems we may encounter in the development of this project as well as the different options that are available with their benefits and inconveniences to make the best choice.

In this chapter an analysis of the server needs, the features that it must provide and its architecture are presented and discussed. A context of the project is also exposed. At the end of the chapter, some remarks on a TDM session at *CHUV* are presented.

2.1 Context of the project

There is a software named *Tucuxi*. This program allows to interpret samples of drug concentrations in a person's blood. It contains a lot of computational functionalities to:

- evaluate the drug concentration normality
- predict a future drug concentration
- suggest a drug dosage adaptation

It also provides a modern GUI with a natural workflow to facilitate the use for any health practitioner and give a clearer feedback to the patient thanks to drug concentration prediction curves and percentiles curves.

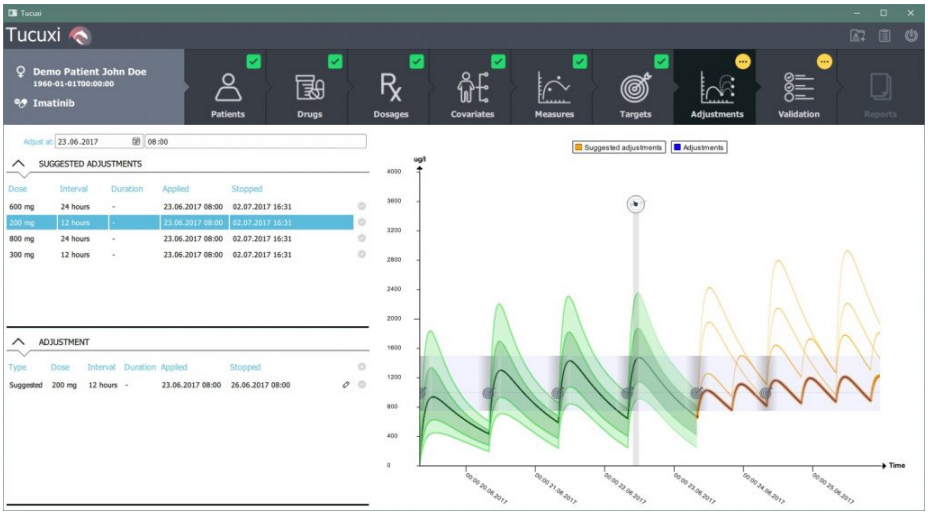


Figure 2.1: GUI of the *Tucuxi* software

Thanks to this program, any health practitioner can perform some *TDM*. It allows them to have fewer intermediaries between the time when the sample is taken and the time when the results must be explained to the patient. The results can also be clearer for the health practitioner as well as the patient thanks to the GUI provided by the software.

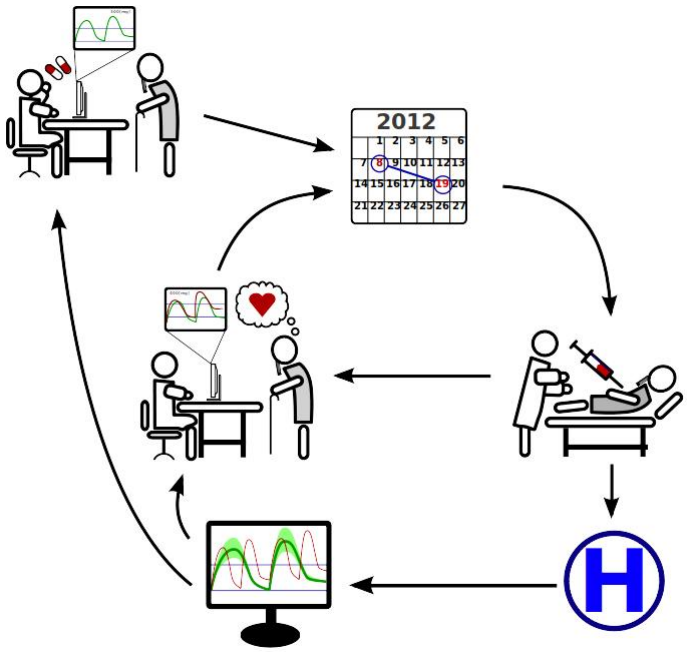


Figure 2.2: Use case of the *Tucuxi* software

This program is made for the medical community as a whole to be used. Not only for the pharmacologists.

2.2 Goal of the project

The goal of this project is to add a server layer to the *Tucuxi* software. This server is be an Expert System. It allows a user to ask it a predefined set of questions and receive an answer regarding a drug concentration prediction, the risk of the actual drug dosage, a new drug dosage or a dug dosage adaptation for a patient.

A server providing this type of service is useful both to the pharmacologists and the other health practitioners. It can make them gain a lot of time. The health practitioners no longer have to wait for the pharmacologist to analyze the blood sample and can directly send a query to the server. having an answer in the following second most of the time. This also means that the patient can receive a feedback from his blood sampling more quickly.
The pharmacologists will probably receive less requests for TDM from the other health practitioners because the answer from the server is enough for them to decide the treatment of the patient. This means that the pharmacologists can use this earned time on the other aspects of their work.

Nothing was done for the server layer before the beginning of this project, so the different features of the server, its architecture, the communication between the client and the server, the problems it could encounter or any other topic that should be discussed at the creation of a server were done from scratch.

2.3 Expert System's features

There server must be able to answer to a predefined set of questions. The answers it will be able to give are listed below:

- **Give a prediction**: Generates a drug concentration prediction
- **Give a first dosage**: Gives a list of suggestions for a first dosage
- **Judge the risk of a dosage**: Indicates if the dosage prescribed puts the patient in an insufficiency/overdosed state or not
- **Judge sample correctness**: Indicates if the last sample is likely to be correct or not
- **Give a dosage adjustment**: Gives a list of suggestions of new dosages that better correspond to the patient's actual condition
- **Tell when a new sample should be taken**: Gives a future date/hour at which a sample should be taken
- **Back extrapolation**
  - **Search for a dosage**: Finds the patient's dosage
  - **Search for the date the drug was ingested**: Finds the date at which the patient last took his medication
- **Report**: Generates a report in PDF format with information from multiple features above

Except for the back extrapolation, these features are requests typically sent to a pharmacologist for a TDM. Back extrapolation is a particular request that doesn't have all the information about the patient's treatment. For example, we don't know what is his dosage, but we do have a blood sample, so we ask the pharmacologist to find what is his dosage to know how much often he must take the drug and the dose of the drug.
The other features give a prediction of the drug concentration in the patient's blood or indicate if the value of the blood sample is in the expected target range, given the patient's health condition.

2.4 Architecture

The sub-sections below show the discussion of the different architecture choices.

2.4.1 REST server

A REST architecture was chosen for the server layer of the *Tucuxi* software. This allows to separate the different features of the server in different URIs.
The REST architecture recommends to use a stateless communication protocol. Stateless means that any request received by the server is not dependent on a previous request, which means that it is possible to scale up or down easily the number of servers that process the requests. In our case we use the HTTP protocol.
With the use of a stateless protocol, the REST architecture aims to be responsive, reliable and scalable. This means that the server will always answer the same thing if we send the same dataset and that it can be scaled up or down depending on the request load on the servers. This is perfect if we want to be able to always answer quickly even if the number of requests grows abnormally.

Stateless versus Stateful

The problem in the case of a stateless application, is that it could take a lot of time to process multiple requests for the same patient. However a stateful application is hard to test and therefore to certify.
So it was decided to add a cache to the server in which the patient's data will be stored for a given time or until the cache is full. This means that we still always send all the data to the server, but if the patient's data are already stored in the server's cache, the computation can be faster. This way we can maintain the possibility of scaling the server, while being able to respond fast in the case of multiple requests for the same patient. It also doesn't change the fact that the server will always answer the same thing if the requests sent are the same.

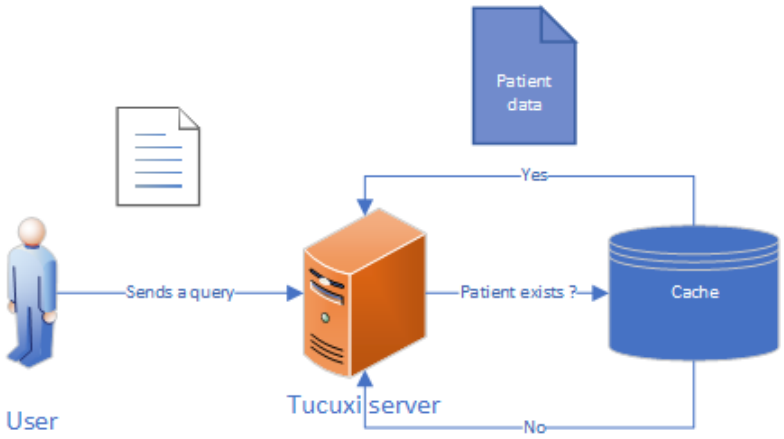


Figure 2.3: Workflow of the solution with a cache

With this method, we must now have a way to know if two patient's data are the same. The first thing we have is the unique identifier of the patient, but this is not enough. For the results of the different requests computations to be relevant, the patient must also have the same covariates. If he doesn't, then he must be considered as a new patient.

To compare two patient's data, two ways are possible:

1. Store the unique identifier and the covariates of the first patient in the cache. Then when we receive a new request, compare these same properties with the ones present in the cache.

2. Compute a hash of the unique identifier and the covariates of the first patient and store it in the cache. Then when we receive a new request, compute the hash of the same properties and compare the two hashes.

However, with the second method, there is a risk for hash collision. We are handling medical data, so we cannot afford to make a mistake on the patient's medical data. To be absolutely sure that the patient is the same, the unique identifier and the covariates of the two sets of data must also be compared after the hashes comparison is valid. The patient's unique identifier and covariates can also be sent back to the client so he can verify that the response is related to the right patient.

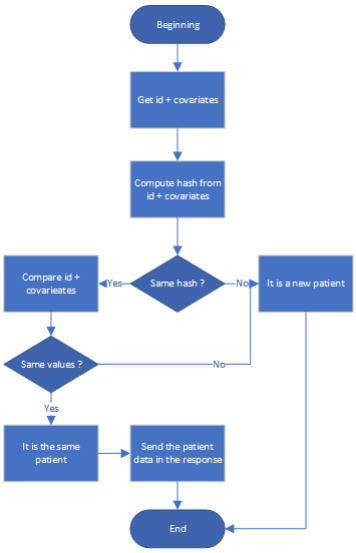


Figure 2.4: Flow chart of the comparison procedure between the patient's data received in the query and the data in the server cache. Version with hashes.

One could argue that the patient's unique identifier and covariates are not enough to compare two patients. In that case the samples can also be added to the comparison. This way the predictions are almost guaranteed to be the same as if we just loaded the new patient's data. The only thing that could potentially make the prediction different is if some new personalized targets were added to the patient's data, but not loaded because the other information are the same.

We still don't know which one of the two comparison methods is the fastest, but it is easily testable in the future.

Centralized endpoint

When we talk about a REST API, the natural question to ask is: what endpoints do we want to provide? Indeed, one of the advantages of the REST architecture is that we can easily separate the features in different URIs. So a natural path would be to make one endpoint for each feature mentioned in the Expert System's features section (2.3). However, what if one day we want to change the REST layer for something else? Depending on what technology we choose, it might take a lot of time to make the modifications.
To overcome this problem, we chose to provide a single computation endpoint, to which all the different requests will be sent. This way, all of the computation and treatment won't change (or just a little bit) if we change the REST layer of the software.

Another advantage of this method is that it allows us to have only one request format that we can still use if the REST layer is changed for another architecture.

Though a REST API may seem useless with this choice, it still is relevant to create additional endpoints for other server features like:

- request the list of the drugs that the software supports
- add a new drug to the server
- update a drug that the server supports
- show the logs of the server
- etc

The Cloud option

When we talk about servers today, it is hard not to mention the Cloud option. Putting a server in the Cloud offers a lot of advantages. Here are some of them:

- The infrastructure is managed by the Cloud service provider
- It is easy to update the software for every client
- The Cloud service providers offer monitoring tools that help us decide if we should scale up or down the number of server dynamically. Which also means a better cost management

As we can see, the Cloud has a lot to offer.
The second point is relevant if there are at least a few different clients, but it is good to envision such a future for the software.
The third point is relevant only if there are automatic requests made by the clients. Otherwise there wouldn't be enough requests for this to be useful.

Even though this software processes medical data, the Cloud service providers guarantee that the virtual machines we rent are secure. However hospitals and government may not want to externalize the medical data for internal or political security reasons. In this case a private server is preferable.

In addition to this, the hospitals seem to have several security layers in their IT infrastructure. If the communication to and from the server must pass through many layers it may slow down the communication.

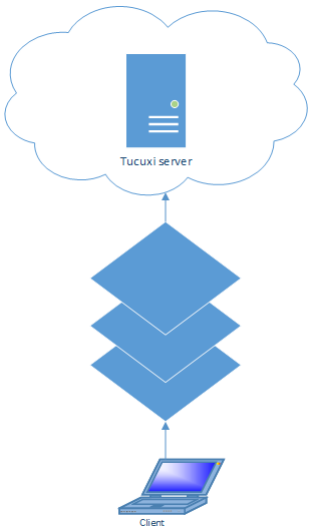


Figure 2.5: Communication between the client and the cloud with the different security layers in between.

Computation speed

One of the main concerns while designing the server's architecture is the response speed. It has been expressed that a response for a request in less than 1 second is preferable.
The first step to accomplish this is to maintain the statelessness of the server (as explained in the stateless versus stateful section (2.4.1)) so that we can scale it up if too many requests are sent.

As seen in the centralized endpoint section (2.4.1), it is possible to send multiple requests at once to the server, so it may not always be possible to respond in less than 1 second to each one.
The solution we envisioned is to have two servers to process the requests.

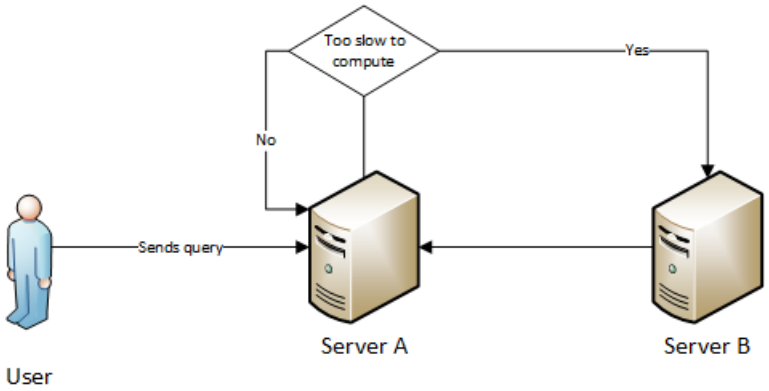


Figure 2.6: Two servers solution

The server A receives all of the requests and decides, according to some criteria, if he can handle them rapidly or not. The criteria could be a certain type of request that is known to be slow, or the number of sub-requests that is too big for example.
If the server finds that the request will take too much time, then he can send the request as is to server B. Server B receives the request and processes it. When he is done, server B sends the response to server A. Server A then proceeds to send this response to the client.

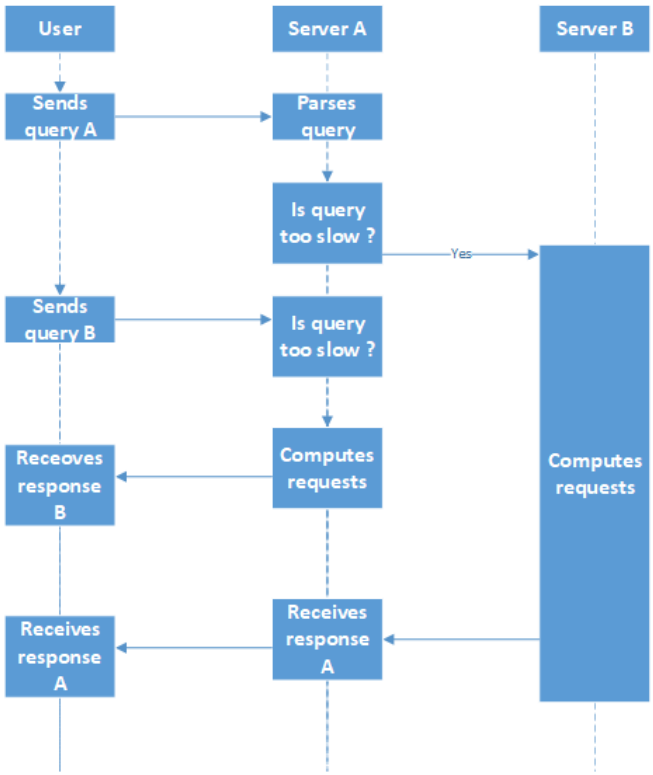


Figure 2.7: Sequence diagram that illustrates the use case of the 2 servers solution

This architecture allows server A to answer rapidly to most of the requests. Without this solution, server A could block for a while on a single request and several other requests would be answered with a lot of delay.

2.4.2 Communication

The communication between the client and the server is done with XML files. This data format was chosen because it was already used in the *Tucuxi* project for the drug description files. It is also a format that the *CHUV* already uses to exchange information with *Tucuxi* from their database (*MOLIS*). *Molis* outputs data in the *HL7* format, which is based on the *XML* format. Then it is translated to *XML* to send it to *Tucuxi*.

In addition to this, the XML format can be verified thanks to an XSD file that describes the structure that the XML file should have. This allows us to verify the query received by the server before retrieving the information present in the query and warn the client that the query is malformed.

2.5 Technologies

The technologies that are used in the functionalities discussed in this report are listed in this section.

2.5.1 Swagger

Swagger is a set of API developer tools that make the API design easier. For this project, we used:

- *Swagger Editor*: allows us to create an API specification using the *YAML* format.
- *Swagger UI*: allows us to visualize the API specification in an interactive UI.
- *Swagger Codegen*: allows to generate a server Stub from the API specification created with the tools mentioned before.

With these tools, we can easily create an API specification. It is also easy to make corrections or to add something to the API.
However the server stub generated for the *Pistache* framework had a problem. It actually generated one *main.cpp* for each endpoint of the specification. This means that this stub was meant to create one server for each endpoint. This means that the URL for each endpoint was different. Some time was taken to modify the code to make it work like a standard REST server, but in the end it was still faster to generate this stub and correct it than to create the server from scratch.

Note that even if the server is generated, we still have to implement all of its functions.

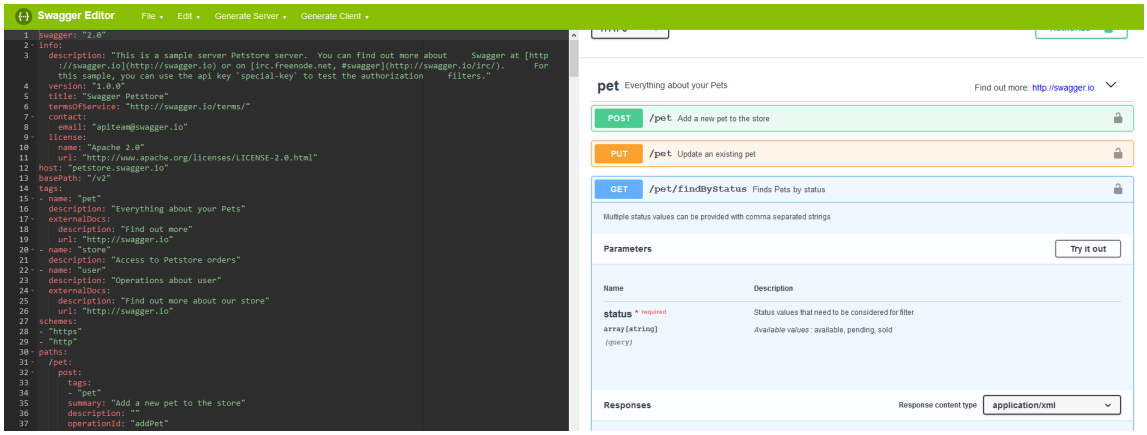


Figure 2.8: Swagger editor and Swagger UI showing an API example

2.5.2 RapidXML

RapidXML is an XML parser written in C++ that focuses on speed. This library allows us to parse the queries sent to the server but also to write answers to these queries in the XML format. This library will also be used to do some Serialization/Deserialization. This step is crucial to test that we correctly retrieve the information sent to the server and for the software certification.

2.5.3 Pistache

Pistache is a C++ framework that allows us to implement an HTTP server or client. We use it only for the server part. It is mainly used to create the HTTP server, receive information via the HTTP requests, send the results of the requests proccessed and setup the REST API endpoints. This framework also has the advantage to be under the *Apache* license. This means that it is completely free and that we can use it for commercial use. It is also easier to certify.

2.6 Remarks

Some questions about the project were brought up after having an encounter with a pharmacologist at *CHUV* and reading some documentation about *Tucuxi*. These questions and remarks are discussed in this section.

2.6.1 Sampling correctness

Some samples are not taken at the stable state. This means that the drug concentration in the blood is still rising or decreasing pretty rapidly. If the sample is not taken at the stable state, then the stable drug concentration could be bigger or smaller than the value present in the blood sample.

The sample must also be plausible. Sometimes the value present in the sample is a lot bigger or a lot smaller than some previous samples or than the population values. In this case the sample could have been taken in a wrong way or the value of the sample was badly written in the database. If such a sample is received by the server, then a particular attention must be given to it for the computation as well as for the software user. Some warning generation could be done to warn the user of the special case of the computation based on this sample.

2.6.2 Dosage correctness

There are a few uncertainties about the doses given to patients. The first one is whether the patient has really taken his drug or not. There are lots of ways for a patient to not take his drug and nurses aren't always there to control this. Sometimes the patient can also vomit the drug after taking it and the nurses don't always realize that he could have thrown up his medication. Depending on his health condition, it could be something that he often does so it isn't an abnormal behaviour for the nurses.
Another problem is that we cannot be sure that the dose has correctly been given to the patient. This is a recurrent problem with very small doses. In particular with children. With very small doses, a difference of 0.1 mg could be pretty big and it is difficult to verify that the nurse that gave the drug to the patient really had this precision.

2.6.3 Covariates correctness

Covariates aren't always given, and when given not always correct. It is recurrent to see the standard weight of the population for example. This probably means that the weight of the patient was not taken and that the health practitioner that sent the request just put a standard value for the weight of the patient.
Some information can also simply be absent or the change between the last value and the present value is too big. It should be possible to send a warning to the user so that he takes it into account when reading the answer from the server.

2.6.4 Information seeking

The biggest problem in the pharmacologist work is to find the information that aren't given by the doctors in other sources. It is the part of the work that is the most time consuming and sometimes the information doesn't exist. The pharmacologist is then forced to make guesses. For this he must find the most plausible case in which the patient could be.

2.6.5 Quantity of doses

Sometimes the pharmacologist could prefer to give more smaller doses to the patient instead of less greater doses, but in reality, some nurses or doctors don't accept to give more than a certain amount of doses per day to the patient. So the pharmacologist resigns to give less greater doses. This means that it is not always the best option for the patient that is used by the health practitioners. The server must be able to suggest more than one option to satisfy this kind of user.

2.6.6 No method switching for dosages

Tucuxi doesn't have a way to switch the dosage method. This means that it isn't possible to change from intrabolus to extrabolus, etc. Nonetheless, a solution to this problem should be implemented in the coming months.

2.6.7 TDM for consultative purpose only

The *TDM*'s work is as consultative purpose only (in *CHUV*). This means that there is no guarantee that the doctors and the nurses respect the proposal of the pharmacologist after his *TDM*. In this case, this means that the responsibility of the dosage administration is given to the health practitioner and not the the *Tucuxi* software.
However the question is still open if the TDM is not for simple consultative purpose.

Chapter 3

Design

Now that we know what the server should be able to do and what it needs to work, we present, in this chapter, a more detailed version of the expert system features, the different endpoints that the server must provide with their respective inputs and outputs and the design of the query format as well as the response format.

3.1 Expert System features

The features listed in the expert system's features section (2.3) are explained and given in more detail in this section.

3.1.1 Give a prediction

Generates a drug concentration prediction. This feature computes a prediction of the drug concentration in the patient's blood on a given date interval. It generates a curve representing this prediction.

Inputs

- the patient's dosage history
- the patient's samples history
- the patients covariates
- a date interval for the start and the end of the prediction
- a list of the wanted percentiles

It will also be possible to ask the following explicit computation:

- population: needs the patient's dosage history
- a priori: needs the patient's dosage history and his covariates
- a posteriori: needs the patient's dosage history, his covariates and his samples history

All of these inputs a optional. All the information available will be used to compute the best case scenario.

Outputs

- a list of tuples "date/value" representing the prediction curve
- a list of tuples "date/value" for each percentile curve requested
- a list of tuples "date/value" representing the samples used for the computation
- a list of cycles that each contain:
  - start and end date
  - a value at the residual state
  - the date and the value at peak
  - the area under the curve (AUC)
  - the mean concentration of the drug
  - if the MIC is present:
    - time over MIC
    - AUC over MIC
    - peak over MIC
    - peak divided by MIC
    - AUC divided by MIC
- a list of the element that were used for the computation
- a text that explains the results
- optional
  - an image in the PNG format of the prediction curve

3.1.2 Give a first dosage

Gives a list of suggestions for a first dosage. This feature provides a list of dosage suggestions to the user without taking into account any former sample. This means that the suggestions are for a first dosage for the patient.

Inputs

- the patient's covariates
- optional
  - a (or some) personalized target(s)

Outputs

- a list of the suggested dosages. Each dosage is associated with a score computed for the given target.
- a text that explains the results
- optional
  - an image in the PNG format of the prediction curve associated with each suggestion

3.1.3 Judge the risk of a dosage

Indicates if the dosage prescribed puts the patient in an insufficiency/overdosed state or not.

Inputs

- the patient's dosage history
  - the history contains the prescribed dosage that we want to evaluate as the last dosage
- the patient's samples history
- the patients covariates
- optional
  - a (or some) personalized target(s)

All of these inputs a optional. All the information available will be used to compute the best case scenario.

Outputs

- indications on the dosage:
  - OK: the concentration prediction is inside the expected range
  - Over/Under-exposed: the concentration prediction is outside of the expected range, but inside the alarm range
  - Danger: the concentration prediction is outside of the alarm range
- a list of the element that were used for the computation
- a text that explains the results
- optional
  - an image in the PNG format of the prediction curve associated with the prescribed dosage

3.1.4 Judge sample correctness

Indicates if the last sample provided is likely or not. Some samples seem to have an absurd drug concentration value. This feature indicates if this value is an error or if it is possible to have such a value.

Inputs

- the patient's dosage history
- the patient's samples history
  - if only 1 sample: is compared with the a priori prediction
  - if 2 or more samples: a flag "likelyhoodUse" is inserted in the samples that we want to use for this computation
- the patients covariates
- optional
  - a (or some) personalized target(s)

Outputs

- indications on the patient's condition:
  - OK: it is very likely that the sample is correct
  - To verify: the sample could be correct, but some elements make it doubtful so it is better to make it verified further
  - Not likely: it is very unlikely that the sample is correct
- the percentile on which the sample is placed. It will be in the range [0, 1]
- a list of the element that were used for the computation
- a text that explains the results

3.1.5 Give a dosage adjustment

Gives a list of suggestions of new dosages that better correspond to the patient's actual condition. Even if the recent samples are inside the target safe values, maybe that the current dosage could be ineffective in the future or even dangerous. This feature allows to verify these assumptions and suggest a list of different dosages for the patient.
The current dosage could also be completely safe, but another dosage could still be better for the patient. For example give less doses per day or a smaller dose could be even better than the current one.

Inputs

- the patient's dosage history
- the patient's samples history
- the patients covariates
- optional
  - a (or some) personalized target(s)

All of these inputs a optional. All the information available will be used to compute the best case scenario.

Outputs

- a list of the suggested dosages. Each dosage is associated with a score computed for the given target. If the actual dosage prediction is inside the expected range of the target, it will have the best score.
- a list of the element that were used for the computation
- a text that explains the results
- optional
  - a loading dose or a resting time. The resting time is a multiple of the ingestion interval
  - an image in the PNG format of the prediction curve associated with the prescribed dosage

3.1.6 Tell when a new sample should be taken

Gives a future date/hour at which a sample should be taken. This feature indicates when a sample should be taken for the patient. Thanks to a drug concentration prediction, the server can see when the peak concentration will be as well as the stable state. With these information, the server can suggest an optimal date/hour for the blood sampling.

Inputs

- the patient's dosage history
- the patient's samples history
- the patients covariates

Outputs

- The date at which the sample should be taken in the String format (or a timestamp from 1.1.1970).
- a text that explains the results

3.1.7 Back extrapolation

Search for a dosage

In this case, we have some values at our disposal like a sample, the patient's covariates and a date at which he took the drug, but we don't know the patient's dosage. The goal of this functionality is to find that dosage.

Inputs

- a sample
- a date of ingestion
- the patients covariates

Outputs

- a list of the possible dosages with each an associated score.
- a text that explains the results
- optional
  - an image in the PNG format of the prediction curve associated with each suggestion

Search for the date the drug was ingested

Here we know the dosage of the patient, but we don't know when he could have ingested his last medication. The goal of this functionality is to find the date at which the patient last took his medication.

Inputs

- a sample
- a dosage
- the patients covariates

Outputs

- The date at which the drug was ingested in the String format (or a timestamp from 1.1.1970).
- a text that explains the results

3.1.8 Report

Returns a PDF file containing the results from all of the requests shown above sent to the server in the same query.

3.2 REST API endpoints

The different REST API endpoints provided by the server are listed and detailed in this section. Each endpoint can have multiple routes. A route is an HTTP call on the same endpoint, but with a different HTTP verb (GET, POST, PUT, DELETE).

3.2.1 /computation

This is the main feature of the server: the expert system. This endpoint allows the user to send requests for a TDM. The server processes them, builds a response and sends it back to the client.

POST Computes the requests sent as input, builds a response and sends it back to the client.

Input: an XML structure following the query structure described in section 3.3.1.
Output: An XML structure with the expected answer from each request sent as input. This answer could also contain a list of errors and warnings. If there are errors, then no computation could be processed by the server. This output is given further details in the communication section (3.3.2).

3.2.2 /drugs

This endpoint allows the user to manage the drugs supported by the server. This means that he can get the list of the drugs supported by the server and add a drug to the list.

GET Fetches the list of all drugs supported by the server.

Returns the "header" node of each XML drug description file that the server is able to manage.

POST Adds a new drug to the list of the ones managed by the server. A special permission is needed to be able to use this endpoint.

Input: an XML description of the drug following the drug description file structure.
Output: An XML structure indicating a success or containing a list of errors.

3.2.3 /drugs/{id}

This is a "subendpoint" of the */drugs* endpoint that focuses on 1 drug that the server already supports. The {id} stands for the unique identifier of the drug.
The user can get information about one specific drug, edit a drug supported by the server or delete one.
If the {id} isn't found within the drug list of the server, a 404 error "Not Found" is sent back to the client.

GET Fetches a drug in the drug list of the server.

Returns the "header" node of the *XML* drug description file that contains the given {id}.

PUT Updates the drug description file containing the given {id} within the drug list of the server. A special permission is needed to be able to use this endpoint.

Input: an XML description of the drug following the drug description file structure.
Output: An XML structure indicating a success or containing a list of errors.

DELETE Deletes the drug description file containing the given {id} from the list of the server. A special permission is needed to be able to use this endpoint.

Returns the number of drugs remaining in the list of the server.

3.2.4 /hello

This is a test endpoint that allows us to see if we can communicate with the server, if it can respond properly and if it can interpret a computation query correctly.

GET Returns the string "Hello World!" to the client

POST Receives a computation query, parses it and returns a field contained in the query to certify that the server could parse the query properly.

Input: an XML structure following the query structure described in section 3.3.1.
Output: One of the fields of the fields contained in the query to certify that the server could parse the query properly.

3.3 Communication

In this section we present the design of the format for the queries that the server will receive and process, as well as the responses that will be sent back to the client.

3.3.1 Query

An XML structure has been designed for the queries sent to the server. To guarantee that this structure is respected, we created an XSD file that describes the format for the queries. Some XSD samples will be used to illustrate the design choices for the queries and some XML samples will be used to illustrate how the query format translates to the actual use case.

Query overview

Here is the broad structure of the query:

```
<xs:element name="query">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="queryID" type="xs:unsignedInt" />
         <xs:element name="clientID" type="xs:unsignedInt" />
         <xs:element name="date" type="xs:dateTime" />
         <xs:element name="language">
            <xs:simpleType>
               <!-- As seen in Tucuxi DataType Enum -->
               <xs:restriction base="xs:string">
                  <xs:enumeration value="en" />
                  <xs:enumeration value="fr" />
                  <xs:enumeration value="de" />
                  <xs:enumeration value="it" />
               </xs:restriction>
            </xs:simpleType>
         </xs:element>
         <xs:element name="admin" type="adminType" minOccurs="0" />
         <xs:element name="parameters" type="parametersType" />
         <xs:element name="requests">
         <xs:complexType>
              <xs:sequence>
                <xs:element name="request" type="requestType"
                                           maxOccurs="unbounded" />
              </xs:sequence>
         </xs:complexType>
      </xs:element>
      </xs:sequence>
```

```
29        <xs:attribute name="version" use="required">
30           <xs:simpleType>
31              <xs:restriction base="xs:string">
32                 <!-- Version number beginning and ending
33                 with a number, each number being
34                 separated by a dot -->
35                 <xs:pattern value="(((\d+)+\.)+)?(\d+)" />
36              </xs:restriction>
37           </xs:simpleType>
38        </xs:attribute>
39     </xs:complexType>
40  </xs:element>
```

The root node is named "query". It has a "version" attribute (line 29) that indicates the version of the query and so the version of the server computation. The "query" node contains the following subnodes: queryID, clientID, date, language, admin, parameters and requests.

- The two first subnodes are for identifying the query and the client. They are mandatory.
- The date indicates the date at which the query was sent to the server. All of the dates in the query and in the response are in the *DateTime* format and respect the following structure: year-month-dayThour:minutes:seconds.

  It is mandatory.
- The language indicates the language that the client wants for the answer that will be sent back. The languages accepted are shown in the enumeration of the XSD above. Each language value in the enumeration respects the ISO 639-1 standard.

  This node is mandatory.
- The "admin" node describes the administrative data of the client, the mandator, their respective personal information and institute information.

  This node is optional.
- The "parameters" node describes the medical data of the patient. It is mandatory.
- The "requests" node is a list of requests structure. The request structure describes the information that the client wants to compute.

  The minimum number of requests in the list is 1, but there is no upper limit.

Here is an example of an xml query overview:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <query version="1.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:noNamespaceSchemaLocation="xml_query.xsd">
5
6     <queryID>123456789</queryID>
7     <clientID>124568</clientID>
8     <!-- Date the xml has been sent -->
9     <date>2018-07-11T13:45:30</date>
10    <language>en</language>
11    <!-- Administrative informations -->
12    <admin>
13       ...
14    </admin>
15    <parameters>
16       ...
17    </parameters>
18    <!-- List of the requests we want the server to take care of -->
19    <requests>
20       <request>
21          ...
22       </request>
23       ...
24    </requests>
25 </query>
```

This structure allows us to separate clearly the different information and their use. The elements of the format are used to identify the query. The administrative data is mostly used for the report generation feature. The parameters are the medical data of the patient that are used for any type of request. The list of requests inform the server of the type of computation that it will perform.

Administrative data

The administrative data is embeded in the "admin" node. Its goal is to provide administrative information about the patient and the mandator of the query. It also embeds some clinical data. It is noteworthy that everything inside the "admin" node as well as the node itself are optional.

Here is the structure of the "admin" node:

```
<xs:complexType name="adminType">
   <xs:sequence>
      <xs:element name="mandator" minOccurs="0">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="person" minOccurs="0">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name="id" type="xs:string" minOccurs="0" />
                        <xs:element name="title" type="xs:string"
                                                 minOccurs="0" />
                        <xs:element name="firstName" type="xs:string"
                                                 minOccurs="0" />
                        <xs:element name="lastName" type="xs:string"
                                                 minOccurs="0" />
                        <xs:group ref="contactGroup" minOccurs="0" />
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
               <xs:element name="institute" type="instituteType" minOccurs="0" />
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="patient" minOccurs="0">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="person" minOccurs="0">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name="id" type="xs:string" minOccurs="0" />
                        <xs:element name="firstName" type="xs:string"
                                                 minOccurs="0" />
                        <xs:element name="middleName" type="xs:string"
                                                 minOccurs="0" />
                        <xs:element name="lastName" type="xs:string"
                                                 minOccurs="0" />
                        <xs:element name="birthdate" type="xs:dateTime"
                                                 minOccurs="0" />
                        <xs:element name="gender" type="xs:string"
                                                 minOccurs="0" />
                        <xs:element name="hospitalStayNumber" type="xs:string"
                                                 minOccurs="0" />
                        <xs:group ref="contactGroup" minOccurs="0" />
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
               <xs:element name="institute" type="instituteType" minOccurs="0" />
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="clinicalData" minOccurs="0">
         <xs:complexType>
            <xs:sequence>
               <xs:any processContents="skip" maxOccurs="unbounded" />
            </xs:sequence>
         </xs:complexType>
      </xs:element>
```

```
58      </xs:sequence>
59  </xs:complexType>
```

The first subnode of the "admin" node is the "mandator" node.

```
1   <xs:element name="mandator" minOccurs="0">
2      <xs:complexType>
3         <xs:sequence>
4            <xs:element name="person" minOccurs="0">
5               <xs:complexType>
6                  <xs:sequence>
7                     <xs:element name="id" type="xs:string" minOccurs="0" />
8                     <xs:element name="title" type="xs:string"
9                                                minOccurs="0" />
10                    <xs:element name="firstName" type="xs:string"
11                                               minOccurs="0" />
12                    <xs:element name="lastName" type="xs:string"
13                                               minOccurs="0" />
14                    <xs:group ref="contactGroup" minOccurs="0" />
15                 </xs:sequence>
16              </xs:complexType>
17           </xs:element>
18           <xs:element name="institute" type="instituteType" minOccurs="0" />
19        </xs:sequence>
20     </xs:complexType>
21  </xs:element>
```

The mandator is defined by some personal information in the "person" subnode and some institute information in the "institue subnode".

The "person" subnode contains the following information: the mandator unique identifier (AVS number), his title, first name, last name, address, phone number and email. These are all personal information. All of the similar information from the mandator's institute must be given in the "institute" subnode.
The "contactGroup" element shown at line 14 contains the definition of the address, phone number and email formats.

The "institute" subnode is defined as follows

```
1   <xs:complexType name="instituteType">
2      <xs:sequence>
3         <xs:element name="id" type="xs:string" minOccurs="0" />
4         <xs:element name="name" type="xs:string" minOccurs="0" />
5         <xs:group ref="contactGroup" minOccurs="0" />
6      </xs:sequence>
7   </xs:complexType>
```

It is defined by a unique identifier of the institute, its name, address, phone number and email address. The phone number and email address could be the phone number and email address of the mandator in this particular institute.

The second subnode of the "admin" node is the "patient" subnode.

```
1   <xs:element name="patient" minOccurs="0">
2      <xs:complexType>
3         <xs:sequence>
4            <xs:element name="person" minOccurs="0">
5               <xs:complexType>
6                  <xs:sequence>
7                     <xs:element name="id" type="xs:string" minOccurs="0" />
8                     <xs:element name="firstName" type="xs:string"
9                                                  minOccurs="0" />
10                    <xs:element name="middleName" type="xs:string"
11                                                 minOccurs="0" />
12                    <xs:element name="lastName" type="xs:string"
13                                                 minOccurs="0" />
14                    <xs:element name="birthdate" type="xs:dateTime"
15                                                 minOccurs="0" />
16                    <xs:element name="gender" type="xs:string"
17                                                 minOccurs="0" />
```

```
                    <xs:element name="hospitalStayNumber" type="xs:string"
                                                          minOccurs="0" />
                    <xs:group ref="contactGroup" minOccurs="0" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="institute" type="instituteType" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

This node is almost the same as the "mandator" node. The only difference is in his personal information. The patient's personal information are: his unique identifier (AVS number), firstname, middle name, last name, birthdate, gender, hospital stay number, address, phone number and email address.
These information are useful for the health practitioner when he generates a report. It allows him to directly have all of the administrative information about the patient directly on the report, saving him the need to check these information on another software or file.

The last subnode of the "admin" node is the "clinicalData" node.

```
<xs:element name="clinicalData" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:any processContents="skip" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

This node is simply a list of anything and of any kind of data.

Here is an example of the administrative data in an actual XML query:

```
<mandator>
      <!-- Mandator personal informations -->
      <person>
        <id>asdf</id>
        <title>Dr.</title>
        <firstName>John</firstName>
        <lastName>Doe</lastName>
        <address>
          <street>Av. de l Ours 2</street>
          <postCode>1010</postCode>
          <city>Lausanne</city>
          <state>Vaud</state>
          <country>Suisse</country>
        </address>
        <phone>
          <number>0213140002</number>
          <type>work</type>
        </phone>
        <email>
          <address>john.doe@chuv.com</address>
          <type>professional</type>
        </email>
      </person>
      <!-- Mandator institute informations -->
      <institute>
        <id>456789</id>
        <name>CHUV</name>
        <address>
          <street>Av. de l Ours 1</street>
          <postCode>1010</postCode>
          <city>Lausanne</city>
          <state>Vaud</state>
          <country>Suisse</country>
        </address>
        <phone>
          <number>0213140001</number>
          <type>work</type>
        </phone>
```

```
39          <email>
40              <address>info@chuv.com</address>
41              <type>professional</type>
42          </email>
43      </institute>
44  </mandator>
45  <!-- Informations relative to the patient -->
46  <patient>
47      <!-- Patient personal informations -->
48      <person>
49          <id>123456</id>
50          <firstName>Alice</firstName>
51          <middleName></middleName>
52          <lastName>Aupaysdesmerveilles</lastName>
53          <birthdate>1970-01-01T00:00:00</birthdate>
54          <gender>Woman</gender>
55          <hospitalStayNumber>1337</hospitalStayNumber>
56          <address>
57              <street>Av. d Ouchy 27</street>
58              <postCode>1006</postCode>
59              <city>Lausanne</city>
60              <state>Vaud</state>
61              <country>Suisse</country>
62          </address>
63          <phone>
64              <number>0216170002</number>
65              <type>work</type>
66          </phone>
67          <email>
68              <address>alice.apdm@gmail.com</address>
69              <type>private</type>
70          </email>
71      </person>
72      <!-- Patient institute informations -->
73      <institute>
74          <id>456789</id>
75          <name>CHUV</name>
76            <address>
77              <street>Av. de l Ours 1</street>
78              <postCode>1010</postCode>
79              <city>Lausanne</city>
80              <state>Vaud</state>
81              <country>Suisse</country>
82            </address>
83          <phone>
84              <number>0213140001</number>
85              <type>work</type>
86          </phone>
87          <email>
88              <address>info@chuv.com</address>
89              <type>professional</type>
90          </email>
91      </institute>
92  </patient>
93  <clinicalData>
94      <date>2018-07-11T13:45:30</date>
95      <name>lastcreatinine</name>
96      <diagnosis>lorem ipsum</diagnosis>
97      <toxicity>lorem ipsum</toxicity>
98      <indication>lorem ipsum</indication>
99      <response>lorem ipsum</response>
100     <value>80.0</value>
101 </clinicalData>
102 </admin>
```

Parameters

The parameters represent the patient's medical data. Its goal is to provide medical data such as the patient's covariates and a list of information for different drugs like the active principle of the drug, the related blood samples, the dosage history and possibly some personalized targets.

Here is the structure of the parameters in the query:

```
<xs:complexType name="parametersType">
   <xs:sequence>
      <xs:element name="patient">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="covariates">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name="covariate" type="covariateType"
                                                     maxOccurs="unbounded" />
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="drugs">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="drug" type="drugType" maxOccurs="unbounded" />
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
```

The "parameters" subnode is divided in two subnodes: patient and drugs.
In the "patient" subnode we find a list of the patient's covariates. These elements are not related to the different drugs. This is why the "patient" node was created, so that if some other drug independent information needs to be added in the future it can be added in the "patient" node.

Here is the definition of a covariate:

```
<xs:complexType name="covariateType">
   <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="date" type="xs:dateTime" />
      <xs:element name="value" type="xs:string" />
      <xs:element name="unit" type="xs:string" />
      <xs:element name="dataType">
         <xs:simpleType>
            <!-- As seen in Tucuxi DataType Enum -->
            <xs:restriction base="xs:string">
               <xs:enumeration value="Int" />
               <xs:enumeration value="Double" />
               <xs:enumeration value="Bool" />
               <xs:enumeration value="Date" />
            </xs:restriction>
         </xs:simpleType>
      </xs:element>
      <xs:element name="nature"  minOccurs="0">
         <xs:simpleType>
            <!-- As seen in the MOLIS specification -->
            <xs:restriction base="xs:string">
               <xs:enumeration value="continuous" />
               <xs:enumeration value="discrete" />
               <xs:enumeration value="categorical" />
            </xs:restriction>
         </xs:simpleType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
```

A covariate is defined by the following elements: its name, date, value, unit, datatype and nature. With this structure, any covariate can be provided to the server.
Some drugs need a particular set of covariates to do some computation. Sometimes a covariate is mandatory, and sometimes the computation can still be done without it. However, each drug can have any kind of covariate and it can be anything from the age, the weight to the presence of a particular tumor. So the structure of a covariate had to be as flexible as possible.

Note that the date of the covariate is the date at which the covariate has been retrieved.
Here is an example of the patient data in the parameters:

```
<patient>
    <covariates>
      <covariate>
         <name>birthdate</name>
         <date>2018-07-11T10:45:30</date>
         <value>1990-01-01T00:00:00</value>
         <unit></unit>
            <dataType>Date</dataType>
         <nature>discrete</nature>
      </covariate>
      <covariate>
         <name>bodyweight</name>
         <date>2018-07-11T10:45:30</date>
         <value>70</value>
         <unit>kg</unit>
            <dataType>Double</dataType>
         <nature>discrete</nature>
      </covariate>
    </covariates>
</patient>
```

The "drugs" subnode contains a list of the drug related data. It was decided that the query can contain medical data about several drugs instead of just one. This allows the server to process the same type of request for different drugs in one query instead of forcing the user to send the same request and medical data changing only the related drug data.

Here is the definition of a drug:

```
<xs:complexType name="drugType">
   <xs:sequence>
      <xs:element name="drugID" type="xs:string" />
      <xs:element name="activePrinciple" type="xs:string" />
      <xs:element name="brandName" type="xs:string" />
      <xs:element name="atc" type="xs:string" />
      <xs:element name="treatment" type="treatmentType" />
      <xs:element name="samples" minOccurs="0">
         <xs:complexType>
              <xs:sequence>
                <xs:element name="sample" type="sampleType" maxOccurs="unbounded" />
              </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="targets" minOccurs="0">
         <xs:complexType>
              <xs:sequence>
                <xs:element name="target" type="targetType" maxOccurs="unbounded" />
              </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
```

The "drug" structure contains the following elements: the drug unique identifier, its active principle, the brand name, its ATC, its related treatment, its related samples and possibily its related personalized targets.
The four first elements are used to identify the drug and have some information about its main active substance and who produces it. The drug identifier is also used in the requests of the query to link the request to the drug.

The treatment node contains the dosage history of the patient. It could have more information in the future so that is why we didn't directly put the dosage history as a subnode of the "drug" node.
The dosage history is based on its *C++* equivalent in the *Tucuxi* project (see figure 3.1).

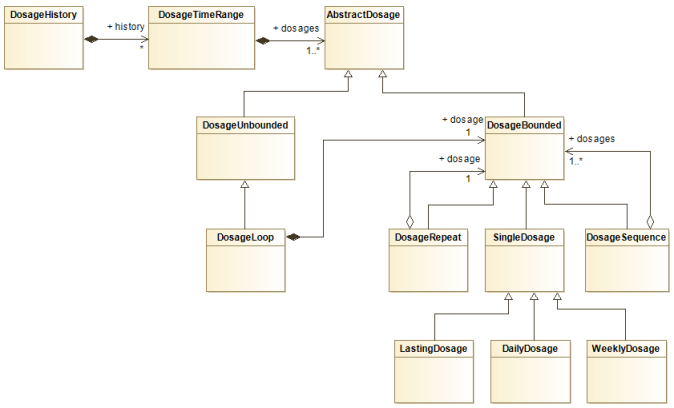


Figure 3.1: UML of the *DosageHistory* class

The structure of the dosage history in the query is made to match its *C++* equivalent as much as possible. To do this, a series of different types were created, each representing a class related to the *C++ DosageHistory* class:

```
<!-- This type and other linked types are made to be the most compatible
with the DosageHistory class and AbstractDosage subclasses -->
<xs:complexType name="dosageHistoryType">
   <xs:sequence>
      <xs:element name="dosageTimeRange" type="dosageTimeRangeType"
                                         maxOccurs="unbounded" />
   </xs:sequence>
</xs:complexType>

<xs:complexType name="dosageTimeRangeType">
   <xs:sequence>
      <xs:group ref="dateIntervalGroup" />
      <xs:element name="dosage" type="dosageType" maxOccurs="unbounded" />
   </xs:sequence>
</xs:complexType>

<!-- Is the equivalent as the AbstractDosage class -->
<xs:complexType name="dosageType">
   <xs:choice>
      <xs:element name="dosageLoop" type="dosageLoopType" />
      <xs:group ref="dosageBoundedGroup" />
   </xs:choice>
</xs:complexType>

<xs:complexType name="dosageLoopType">
  <xs:sequence>
    <xs:group ref="dosageBoundedGroup" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="dosageRepeatType">
   <xs:sequence>
      <xs:element name="iterations">
         <xs:simpleType>
            <xs:restriction base="xs:int">
               <xs:minInclusive value="1" />
            </xs:restriction>
         </xs:simpleType>
      </xs:element>
      <xs:group ref="dosageBoundedGroup" />
   </xs:sequence>
</xs:complexType>

<xs:complexType name="dosageSequenceType">
   <xs:sequence>
      <xs:group ref="dosageBoundedGroup" maxOccurs="unbounded" />
   </xs:sequence>
```

```
</xs:complexType>

<xs:complexType name="lastingDosageType">
   <xs:sequence>
      <xs:element name="dateInterval" type="dateIntervalType" />
      <xs:group ref="intakeGroup" />
   </xs:sequence>
</xs:complexType>

<xs:complexType name="dailyDosageType">
   <xs:sequence>
      <xs:element name="time" type="xs:time" />
      <xs:group ref="intakeGroup" />
   </xs:sequence>
</xs:complexType>

<xs:complexType name="weeklyDosageType">
   <xs:sequence>
      <xs:element name="day">
         <xs:simpleType>
            <xs:restriction base="xs:int">
               <xs:minInclusive value="0" />
               <xs:maxInclusive value="6" />
            </xs:restriction>
         </xs:simpleType>
      </xs:element>
      <xs:element name="time" type="xs:time" />
      <xs:group ref="intakeGroup" />
   </xs:sequence>
</xs:complexType>
```

With this structure for the "dosageHistory" node, we made sure that it is possible to instatiate a *DosageHistory* class.

Here is an example of a treatment structure in an XML query:

```
<treatment>
   <dosageHistory>
      <dosageTimeRange>
         <start>2018-07-06T13:45:30</start>
         <end>2018-07-12T13:45:30</end>
         <dosage>
            <dosageLoop>
               <weeklyDosage>
                  <day>1</day>
                  <time>12:00:00</time>
                  <dose>
                     <value>250</value>
                     <unit>mg</unit>
                  </dose>
                  <formulationAndRoute>
                     <formulation>ParenteralSolution</formulation>
                     <administrationName>foo bar</administrationName>
                     <administrationRoute>IntravenousBolus</administrationRoute>
                     <absorbtionModel>Intravascular</absorbtionModel>
                  </formulationAndRoute>
               </weeklyDosage>
            </dosageLoop>
         </dosage>
      </dosageTimeRange>
   </dosageHistory>
</treatment>
```

This example represents a dosage that was given from the 07-06-2018 at 13:45:30 to the 07-12-2018 at 13:45:30. It is a weekly dosage that is given every Monday at 12:00:00. The dose is of 250 mg. The administration route of the the drug is intravenous bolus and the absorbtion model is intravascular.

After the "treatment" node comes the "samples" node. This is a list of a sample definition:

```
<xs:complexType name="sampleType">
```

```
    <xs:sequence>
       <xs:element name="sampleID" type="xs:positiveInteger"/>
       <xs:element name="sampleDate" type="xs:dateTime" />
       <xs:element name="arrivalDate" type="xs:dateTime" />
       <xs:element name="concentrations">
             <xs:complexType>
               <xs:sequence>
                   <xs:element name="concentration" maxOccurs="unbounded">
                      <xs:complexType>
                      <xs:sequence>
                            <xs:element name="analyteID" type="xs:string" />
                            <xs:element name="value" type="xs:decimal" />
                            <xs:element name="unit" type="xs:string" />
                      </xs:sequence>
                      </xs:complexType>
                </xs:element>
               </xs:sequence>
             </xs:complexType>
       </xs:element>
       <xs:element name="likelyhoodUse" type="xs:boolean" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
```

With the sample structure we have all of the information about a blood sample. The first node is the sample identifier, then comes the date at which the sampling was done and the date at which the sample was received in the laboratory.
A drug blood sample can be related to multiple analytes. To take this matter into account, we created a "concentrations" node that lists the different analytes of the sample and their related value and unit.
In addition to this, we added a "likelyhoodUse" node. This node is used for the *Judge sample correctness* expert system feature. The user can set this node to *True* or *False* depending on wether the sample should be used to judge if the last sample is abnormal or not.

Here is an example of a sample structure in an XML query:

```
<sample>
   <sampleID>123456</sampleID>
   <sampleDate>2018-07-08T20:00:00</sampleDate>
   <arrivalDate>2018-07-08T22:00:00</arrivalDate>
   <concentrations>
      <concentration>
         <analyteID>vancomycin</analyteID>
         <value>10.0</value>
         <unit>mg</unit>
      </concentration>
   </concentrations>
</sample>
```

The las node of the drug structure is the "targets" node. This node is a list of personalized targets for the patient.

```
<xs:complexType name="targetType">
   <xs:sequence>
      <xs:element name="activeMoietyID" type="xs:string" />
      <xs:element name="targetType">
         <xs:simpleType>
            <!-- This enumeration is based on the one in the drug file XSD -->
            <xs:restriction base="xs:string">
                  <xs:enumeration value="peak" />
                  <xs:enumeration value="residual" />
                  <xs:enumeration value="mean" />
                  <xs:enumeration value="auc" />
                  <!-- To use the enumerations below MIC should be a covariate -->
                  <xs:enumeration value="aucOverMic" />
                  <xs:enumeration value="timeOverMic" />
                  <xs:enumeration value="aucDividedByMic" />
                  <xs:enumeration value="peakDividedByMic" />
              </xs:restriction>
         </xs:simpleType>
```

```
19          </xs:element>
20          <xs:element name="unit" type="xs:string" />
21          <xs:element name="inefficacyAlarm" type="xs:decimal" />
22          <xs:element name="min" type="xs:decimal" />
23          <xs:element name="best" type="xs:decimal" />
24          <xs:element name="max" type="xs:decimal" />
25          <xs:element name="toxicityAlarm" type="xs:decimal" />
26      </xs:sequence>
27  </xs:complexType>
```

The target contains the following information: the active moiety identifier, the target type and the values for the different ranges of the target with the related unit.

The target type means at what point in the drug concentration curve do we compare the range values to.

There are two ranges of values for the target. The first range is the safe range. It is represented by the "min" and "max" values. If the analyzed drug concentration is within this range, the patient is in a safe state. The "best" value is the value that the drug concentration must aim at.
The second range is the alarm range. It is represented by the "inefficacyAlarm" and the "toxicityAlarm" values. If the analyzed drug concentration value is outside of the safe range, but inside the alarm range, then the drug could be inefficient, but we could still calmly change the dosage to make the drug concentration inside the safe range.
However if the analyzed drug concentration value is outside of the alarm range, then it is urgent to take some measures and change the dosage of the patient.

Here is an example of a target structure in an XML query:

```
1  <target>
2          <activeMoietyID>vancomycin</activeMoietyID>
3          <targetType>residual</targetType>
4          <unit>mg</unit>
5            <inefficacyAlarm>15</inefficacyAlarm>
6            <min>20</min>
7            <best>25</best>
8            <max>30</max>
9            <toxicityAlarm>50</toxicityAlarm>
10     </target>
11 </targets>
```

Requests

The last subnode of the "query" node is the "requests" node. It is a list of request structures. The goal of a request structure is to provide information to the server that allows it to know to what drug the request is related and what kind of computation it must process.
It was decided that several requests can be sent to the server at the same time. This is why the "requests" node is a list of request structures. It allows the server to use the same medical data of the patient for all of the different requests received instead of receiving several requests that contain the exact same medical data for the same patient. With this solution we gain some time on the communication between the client and the server, but also at the parsing of the query that is done only once.

The request structure is as follows:

```
1  <xs:complexType name="requestType">
2     <xs:sequence>
3        <xs:element name="requestID" type="xs:string" />
4        <xs:element name="drugID" type="xs:string" />
5        <xs:element name="requestType">
6           <xs:simpleType>
7              <xs:restriction base="xs:string">
8                    <xs:enumeration value="prediction" />
9                    <xs:enumeration value="firstDosage" />
10                   <xs:enumeration value="dosageRisk" />
11                   <xs:enumeration value="dosageLikelyhood" />
12                   <xs:enumeration value="dosageAdaptation" />
```

```
13                   <xs:enumeration value="sampleDate" />
14                   <xs:enumeration value="backextrapolation:dosageSearch" />
15                   <xs:enumeration value="backextrapolation:dateSearch" />
16                   <xs:enumeration value="sampleDate" />
17                   <xs:enumeration value="report" />
18               </xs:restriction>
19            </xs:simpleType>
20         </xs:element>
21         <xs:element name="dateInterval" type="dateIntervalType" minOccurs="0" />
22         <xs:element name="predictionType" minOccurs="0">
23            <xs:simpleType>
24               <xs:restriction base="xs:string">
25                  <xs:enumeration value="population" />
26                  <xs:enumeration value="a priori" />
27                  <xs:enumeration value="a posteriori" />
28                  <xs:enumeration value="best" />
29               </xs:restriction>
30            </xs:simpleType>
31         </xs:element>
32         <xs:element name="graph" type="graphType" minOccurs="0" />
33         <xs:element name="percentiles" type="percentilesType" minOccurs="0" />
34         <xs:element name="backextrapolation" minOccurs="0">
35            <xs:complexType>
36               <xs:choice>
37                  <xs:group ref="BE_dosageSearchGroup" />
38                  <xs:group ref="BE_dateSearchGroup" />
39               </xs:choice>
40            </xs:complexType>
41         </xs:element>
42      </xs:sequence>
43   </xs:complexType>
```

The request structure contains the following information: the request unique identifier, the drug unique identifier related to the request, the request type, a date interval, a prediction type, the request for a graph, the percentiles that should be computed in addition to the the main curve and a special subnode for the back extrapolation feature.
Only the three first elements are mandatory. Default values are used if the other elements are not given.

The drug unique identifier is used by the server to retrieve the medical data related to the good drug. At first we thought about putting a "requests" node inside the drug structure, but it was preferable to have all of the requests at the same place. It also made some sense that the requests defined the query instead of defining the drug data.

The request type indicates what type of computation, that is what expert system feature is requested by the user. The request type is an enumeration and a new entry can be added in case a new feature is added to the expert system.

The date interval of the request represents the start date and the end date of the computation. This interval's maximum time is 31 days in the default server configuration. More than 1 month could break the server because it computes to much points in the prediction curves. Especially if it also computes percentiles.

The prediction type is a way for the user to explicitly ask to the server to do a certain type of prediction:

- population: is normally done when we do not have any medical data about the patient except for his dosage history
- a priori: is normally done when we have the patient's covariates, but no samples
- a posteriori: is normally done when we have the patient's covariates and blood samples
- the best prediction type given the medical data of the patient

The graph node is used to ask for a PNG image of the prediction curve in addition to the standard information computed by the server.
The graph request can be personalized:

```
<xs:complexType name="graphType">
   <xs:sequence>
      <xs:element name="dateInterval" type="dateIntervalType" />
      <xs:element name="percentiles" type="percentilesType" />
   </xs:sequence>
</xs:complexType>
```

The graph date interval can be different than the interval of the request and the user can also ask for fewer or more percentile curves on the graph.

The percentiles node is used to ask for several additional curves to compute on the provided percentiles. The percentiles node is a list of numbers ranging from 0 to 100.
Thanks to the percentile curves the user can better interpret the sample of the patient and the results given by the server. It can help him see where the patient is placed in comparison to the population.

Finally the back extrapolation node is a special node. The back extrapolation feature needs special information about the patient so they are given directly in the request.
The back extrapolation node can contain either of two data sets:

- A dataset containing only a sample. This is for the *Search for a dosage* feature. This feature takes a sample and tries to find the actual dosage of the patient.
- A dataset containing a sample without sampling date and a dosage. This is for the *Search for the date the drug was ingested* feature. This feature tries to find at which date the drug of the dosage was taken given the incomplete sample.

Here is an example of a request:

```
<request>
   <requestID>123abc</requestID>
   <drugID>vancomycin</drugID>
   <requestType>dosageAdaptation</requestType>
   <dateInterval>
      <start>2018-07-06T13:45:30</start>
      <end>2018-07-12T13:45:30</end>
   </dateInterval>
   <predictionType>best</predictionType>
   <graph>
      <dateInterval>
         <start>2018-07-11T13:45:30</start>
         <end>2018-07-12T13:45:30</end>
      </dateInterval>
      <percentiles>
         <percentile>50</percentile>
      </percentiles>
   </graph>
   <percentiles>
      <percentile>75</percentile>
      <percentile>50</percentile>
      <percentile>25</percentile>
   </percentiles>
</request>
```

With this query structure we have all the information needed for the different expert system features and even more. However these addition information could be of use in future development if some new feature can make use of them.

The complete XSD that allows to verify this XML format as well as a complete XML example can be found on this GitLab repository [1].

[1] https://gitlab.com/REDS-tdb/2018-benallal/tree/master/dev/xml_query

3.3.2 Response

The user expects a response after sending a query to the server. This response must contain the results of the expert system's computation, but it must also contain some warning or some errors encountered during the computation or the parsing of the query.

Here is an example of response:

```
<responses>
   <response>
      <queryID>1234</queryID>
      <requestID>123456</requestID>
      <issues>
         <status>Error</status>
         <errors>
            <error>
               <id>anerrorid</id>
               <message lang="en">Node query/parameters/drugs/drug/target/min
               contains 'string' instead of 'decimal'</message>
            </error>
            <error>
               <id>anothererrorid</id>
               <message lang="en">'vancomy': Unknown drugID</message>
            </error>
         </errors>
         <warnings>
            <warning>
               <id>awarningid</id>
               <message lang="en">'query/parameters/drugs/drug/target/unit':
               Node is empty</message>
            </warning>
         </warnings>
      </issues>
      <data>
         ...
      </data>
   </response>
   <response>
      ...
   </response>
   ...
</responses>
```

The response sent back to the client is a list of response structures. One for each request present in the query received by the server. This may duplicate the warnings and the errors, but it is easier to let the client decide if he wants to process each request separately or if he wants to process the response as a whole.

The two first nodes of a response is the query unique identifier and the request unique identifier. This allows the client to know exactly to which query and request this response is related to.

These first two nodes are followed by a "issues" node. This node contains the status of the computation, a potential list of errors and a potential list of warnings.
The status can have one of two values: Success or Error. The "Success" value is written if the computation was able to be processed without error. The "Error" value is written if an error is encountered while parsing the query or during the computing of the request. "Error" means that the computing of the request could not be done.

The list of errors indicates all the error encountered during the parsing of the query or during the computing of the request. If an error is encountered, then the computation cannot finish. An error is structured as follows: an error unique identifier and the message of the error.
The error unique identifier is used to pick a standard error in a file containing these standard errors. Here is an example of this file:

```
<errors>
   <error>
      <id>empty node</id>
```

```
        <message lang="en">Node is empty</message>
    </error>
    <error>
        ...
    </error>
    ...
</errors>
```

This way it is easier to add new errors to the program.

The message of the error contains the actual meaning of the error. It also has an attribute to indicate the language of the message content. As for the language node of the query, this attribute respects the ISO 639-1 norm.

After the error list is the warning list. This list is structured the exact same way as the errors, but it has another meaning. A warning is something that the server was not expecting in the query or during the computation of the request, but did not prevent it to successfully finish the computation.
The warning also have a file containing standard warning messages:

```
<warnings>
    <warning>
        <id>unknown_drug_id</id>
        <message lang="en">Unknown drugID</message>
    </warning>
    <warning>
        ...
    </warning>
    ...
</warnings>
```

Finally the response contains the actual data resulting from the computation of the request. This data is structured differently depending on what type of request was sent to the server. The client can use the request unique identifier provided in the response to know what request type was asked and read the data of the response accordingly.

Chapter 4

Implementation

In this chapter we present how we implemented the server and the different elements presented in the *Design* chapter.

4.1 Files organization

Here we present how *Tucuxi* server project files are organized.

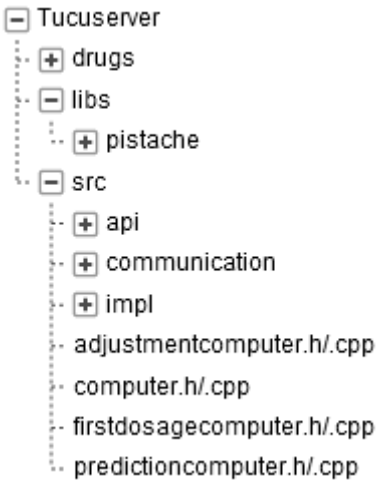


Figure 4.1: Directory tree of the *Tucuxi* server project

The *Tucuserver* root folder contains the following sub-folders: drugs, libs and src.
The drugs folder contains the different drug description files that allow us to instantiate a *DrugModel* object.
The libs folder contains the libraries used for the project. Only the *Pistache* library is used currently.
The src folder contains the source code of the server. Directly under the src folder we have the *main.cpp* file as well as the computing classes that process the requests sent to the server. The main file loads some parameters in a configuration file and starts the REST server that will wait for the requests.
The src folder contains 3 subfolders: api, communication and impl.

- api: This folder contains the classes that allow to setup the different endpoints and routes of the server. More details are given in the REST API section (4.3).
- communication: This folder contains the classes used for serialization. The *XMLReader* class is used to create a *Query* object that contains all the information about the patient as well as the requests to process.

There is also a response class that is constructed while processing the different requests. This class is then serialized to send the results of the requests. More details about serialization are given in the serialization/deserialization section (4.4).

- impl: This folder contains the implementation of the *api* folder. The methods of these classes are called by the ones in the *api* folder and implement the behaviour that the corresponding endpoint awaits. More details are given in the REST API section (4.3).

4.2 Server configuration

Some parameters of the server need to be possibly changed by the server owner. To achieve this, a configuration file was made. This file contains information that will be read at launch and will persist during all the server's running time.
The information currently used in the configuration file are:

- The port of the URI for the REST API
- The number of points computed for each cycle
- The maximum date interval of the request in days

If a new parameter is needed in future development, it can easily be added to the configuration file.

The configuration file is in the *XML* format. Here is the configuration file so far:

```
<configuration>
    <net>
        <port>9090</port>
    </net>
    <computation>
        <!-- The number of points for each cycle -->
        <cycleSize>250</cycleSize>
        <!-- The maximum interval of the request in days. -->
        <maxDateInterval>31</maxDateInterval>
    </computation>
</configuration>
```

There are two groups of parameters so far: *net* and *computation*.
The *net* group contains all of the parameters needed to setup the REST API and the HTTP communication. For example, we could add the host IP address of the server in the future.
The *computation* group contains all of the parameters needed to setup the computation of the requests sent to the server. For example, we could add a time limit to the processing of a request in the future.
The file parsing is explained in the serialization/deserialization section (4.4.3).

The configuration file is read only at the server launch. To preserve the server's parameters, a *Configuration* class was created. This class is a *Singleton*:

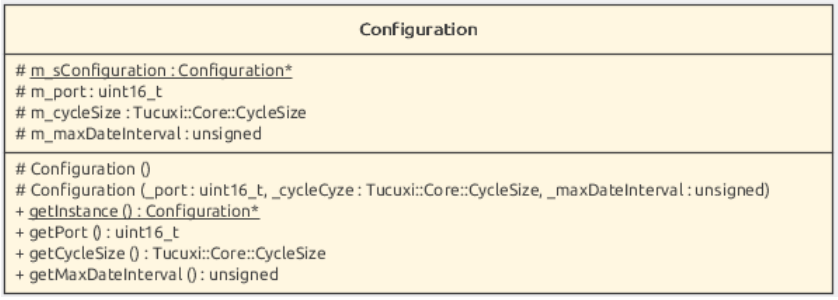


Figure 4.2: UML of the Configuration class

The *Singleton* design pattern makes us instantiate the *Configuration* class only once. This way the same parameters will be used through all of the program, avoiding any inconsistency related to these parameters.

4.3 REST API

For the implementation of the REST layer, we used the *Pistache* framework (2.5.3). The REST API is structured as such:

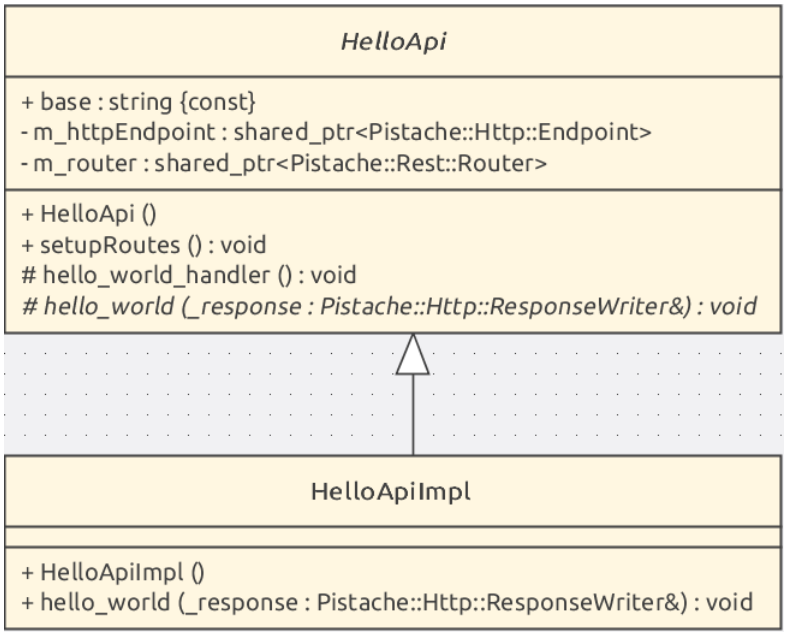


Figure 4.3: UML of the relation between the HelloApi classe and the HelloApiImpl class

The *Api* classes have the role of setting up the different routes of the endpoint and calling the related methods. An *Api* class has *handler* methods and abstract methods. The *handler* methods are used to call the abstract method and send a message back to the client if an exception is caught in the process. The abstract methods implement the actual behaviour expected by a call on the corresponding route.
For example in the HelloApi class, there are two routes:

- GET on */hello*
- POST on */hello*.

The first route is bound to the *hello_world_handler()* method. This method then calls the abstract method *hello_world()* and sends an error message if an exception is caught. This las method just sends the "Hello World!" string to the client. This is the expected behaviour of a call on the */hello* endpoint with an HTTP GET.
Here is the implementation of the *hello_world_handler()* method:

```
1  void HelloApi::hello_world_handler(Request& _request, ResponseWriter _response) {
2      UNUSED(_request);
3      try {
4        this->hello_world(_response);
5      } catch (std::runtime_error&  e) {
6        //send a 400 error
7        _response.send(Pistache::Http::Code::Bad_Request, e.what());
8        return;
9      }
10 }
```

Then there are the *ApiImpl* classes. These classes inherit from the *Api* classes. Their role is to implement the expected behaviour of the superclass abstract methods.
In the example above, the implementation of the *hello_world()* method is written in the *HelloApiImpl* class. Here is its implementation:

```
1 void HelloApiImpl::hello_world(Pistache::Http::ResponseWriter& _response) {
2     _response.send(Pistache::Http::Code::Ok, "Hello World!\n");
3 }
```

This implementation allows us to separate the proccessing of the HTTP request in the *Api* classes and the processing of the query in the *ApiImpl* classes.

4.4 Serialization/Deserialization

The serialization and deserialization were done with the RapidXML framework (2.5.2). A higher level implementation of XML parsing was done in the *Tucuxi* project. This is the actual parser we used in the server for the serialization.
Some typical use of the parser is as such:

```
1         string xml = "some xml content";
2         Common::XmlDocument xmlDocument;
3         if(xmlDocument.fromString(xml)) {
4             Common::XmlNode root = xmlDocument.getRoot();
5             string someValue = root.getChildren(SOME_VALUE_NODE_NAME)->getValue();
6             // Use someValue
7         }
```

The XML parsing is used for three things:

- Transform the query received in the HTTP request
- Transform the response after the processing of the query to send it to the client
- Read a configuration file to set some global parameters at runtime.

Every serialization or deserialization is done by hand.

4.4.1 Query

The query deserialization has one goal: get all the information received in the body of the HTTP request to create a *Query* object that is the only owner of all these information.
The *Query* class corresponds to the XML structure of a query (3.3.1). Most data stored in the class is of a simple type (*int*, *double*, *string*, *DateTime*). The only exception is the dosage history that is directly transformed into a *DosageHistory* object.
This was possible because the *XML* structure of the dosage history was directly taken from its *C++* structure (see figure 3.1).

All the other information are of simple types because it may contain more information than needed to directly create the corresponding object in the *Query* class. For example, the *Sample* class used for computation doesn't need the *arrivalDate* data, but it is present in the *XML* structure of a sample. So we created a *SampleData* class that contains all of the information sent in the *XML* structure of a sample. This class contains simple types that can be used later to create a *Sample* object.

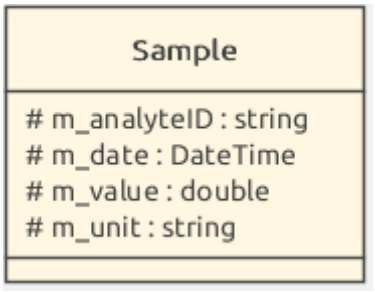


Figure 4.4: UML of the *Sample* class used for computation

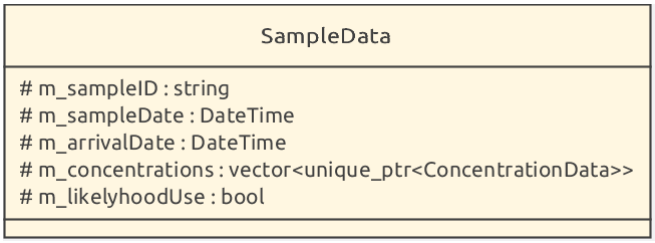


Figure 4.5: UML of the *SampleData* class used to store the sample information received with the query

A lot of of classes were made to store all of the information contained in the query typcially in the administrative data node or in the request node. As mentioned before, the only custom class that was not created to store the related data in the query is the *DrugHistory* class.

4.4.2 Response

For the response, we created some classes to represent the response status, the messages and the issues. The *Response* class must at least composed of these other elements. However, the real purpose of a response is to send the results of the computation to the client. This means that some data must be added to the response.
In addition to this, each feature of the server has a different output. This means that a specific response class for each of the possible outputs must be created.

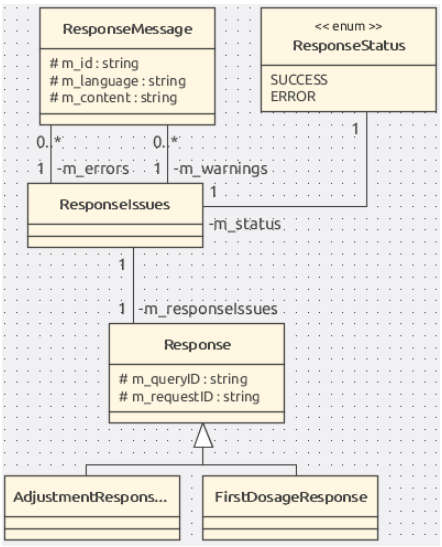


Figure 4.6: UML of the *Response* class and subclasses

Thanks to such an architecture, it is easy to separate the construction of different response type according to the request type.

Unfortunately, the response of the server is not fully implemented. The idea of different *Response* subclasses specialized in building the correct response was done while implementing the *Response* class, but the final result could be achieved.

4.4.3 Configuration

The *Configuration* class is a *Singleton*, which means that it is instantiated only once when the *getInstance()* method is called for the first time. This means that the server's configuration file is read at the first call of the *Configuration*'s *getInstance()* method. To respect these conditions, we created a *ConfigurationReader* class.
This class simply contains the same attributes as the *Configuration* class and their related Getters. This way we can create an instance of the *Configuration* class just after reading the configuration file.
Here is the *getInstance()* method:

```
Configuration* Configuration::getInstance()
{
    if (m_sConfiguration == nullptr) {
        ConfigurationReader configurationReader("some_path/tucuserver.cfg");
        m_sConfiguration = new Configuration(
                    configurationReader.getPort(),
                    configurationReader.getCycleSize(),
                    configurationReader.getMaxDateInterval()
                    );
    }

    return m_sConfiguration;
}
```

Having a *ConfigurationReader* class allows us to separate the parsing of the configuration file and the instantiation of a *Configuration Singleton*. This way, if we want to change the way we parse the configuration file or if the configuration file is not in the *XML* format for example, we only need to change the *ConfigurationReader* class without changing the *Configuration* class at all.

4.5 Computing

After treating the HTTP request and created a *Query* object, we can now use this information contained in the *Query* object to do some computation. To separate the REST API role from the computation role, we created a *Computer* class and a subclass for each request type that the server supports:

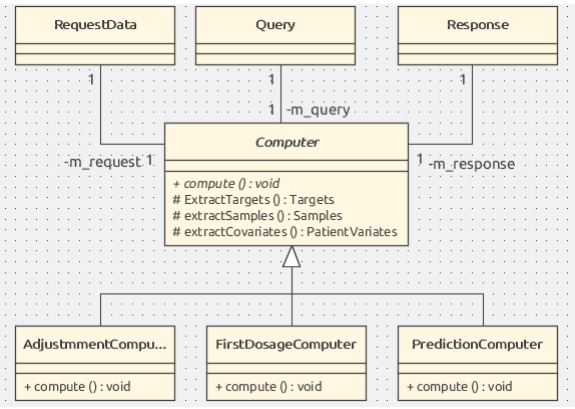


Figure 4.7: UML of the *Computer* class and its subclasses

The principle of the *Computer* class and subclasses is simple: all subclasses must implement their version of the *compute()* method. This method's goal is to have the expected behaviour of the related request type. It constructs a *Response* object while encountering errors and with the result of the computation.
When the *compute()* call is finished, we can get the response constructed and transform it to send the result to the client.

A simple use of a *Computer* is as follows:

```
void ComputationApiImpl::compute_requests(const Query& _query,
                                          ResponseWriter& _response) {

    unique_ptr<Computer> computer = make_unique<AdjustmentComputer>(_query);
    computer->compute();

    // send a string
    _response.send(Pistache::Http::Code::Ok, computer->getResponse().someString());
}
```

This way, the code in the implementation of the REST API is minimal and we can change the REST API layer easily.

The *Computer* class also posses *protected* methods that can be used in its different subclasses. An example is the *extractSamples()* method.

```
Samples Computer::extractSamples(size_t _drugPosition) const
{
    Samples samples;
    const vector< unique_ptr<SampleData> >& samplesData = m_query.getpParameters()
                                                          .getDrugs().at(_drugPosition)
                                                          ->getSamples();

    for (const unique_ptr<SampleData>& sd : samplesData) {
        for (const unique_ptr<ConcentrationData>& cd : sd->getConcentrations()) {
            samples.push_back(
                        make_unique<Sample>(
                            sd->getpSampleDate(),
                            cd->getAnalyteID(),
                            cd->getValue(),
                            Core::Unit(cd->getUnit())
                        )
            );
        }
    }

    return samples;
}
```

This method searches for the sample data corresponding to the drug ID of the request and proceeds with creating *Sample* instances that will be added to a *DrugTreatment* instance used for the computation (4.5.1).
The same type of method was also made for the *Target* class and for the *PatientCovariate* class. These classes are used by the *DrugTreatment* for the computation.

4.5.1 How to compute

We make the computation thanks to the *Tucuxi* project on which this server is based on. This project uses a *ComputingComponent* class to compute the result and this *ComputingComponent* needs a *ComputingRequest* object to process the computation.
A *ComputingRequest* is defined by a request/response ID, drug model, drug treatment and computing trait.

```
IComputingService* component =
        dynamic_cast<IComputingService*>(ComputingComponent::createComponent());

ComputingRequest request(requestResponseID, *drugModel,
                         drugTreatment, computingTraits);
unique_ptr<ComputingResponse> response =
```

```
        make_unique<ComputingResponse>(requestResponseID);

ComputingResult result = component->compute(request, response);
```

The request/response ID is a *String* that allows the client to recognize to which request the response is related.
The drug model is the object representing the drug described in the related drug description file. A drug description file is an *XML* structure that represents the drug data such as: its identifier, name, active substance, related covariates, population targets, etc.
The drug treatment represents the information about the patient's treatment. It contains the dosage history of the patient, his covariates, a list of his blood samples and a list of some personalized targets. All of these data are optional and personalized targets will likely be empty most of the time. Depending on how much information we have about the patient, we can compute different types of predictions:

- the population prediction if we have no information
- the a priori prediction if we only have the patient's covariates
- the a posteriori prediction if we have the patient's covariates and samples

The computing trait is a class that embeds all the information needed to do a certain type of computation. There are several *ComputingTrait* subclasses that can be used to do the right computation.

When all of these elements are ready, the *ComputingComponent* can be used to process the computation and build the *ComputingResponse*. When the computation is done, the *ComputingResponse* will contain a list of *SingleComputingResponse*. This class has several subclasses that we'll retrieve depending on what type of computation, that is what *ComputingTrait* subclass we chose.

```
const vector<unique_ptr<SingleComputingResponse>> &responses =
                                                response.get()->getResponses();

for(size_t i = 0; i < responses.size(); i++) {
    const SpecificResponse* resp =
                        dynamic_cast<SpecificResponse*>(responses[i].get());

    // build the REST API response
}
```

Finally, we can use the list of *SingleComputingResponse* to build the response of our REST API.

4.5.2 Adjustements

To treat the adjustment requests, an *AdjustmentComputer* class was created. This class only overrides the *compute()* method of its superclass: the *Computer* class.
In this method we proceed like follows:

- We retrieve the drug unique identifier of the request we are processing.
- We import the *DrugModel* corresponding to the drug id
- We retrieve the dosage history, the patient's covariates, his samples and his personalized targets in the medical data corresponding to the drug id.
- We build the *DrugTreatment* with the last infromation retrieved.
- We retrieve some information of the request like the request id, its date interval and its prediction type.
- We compute the adjustment time of the request. This part is not done yet, but the aim is to find the adjustment time automatically, depending on the patient's dosage history and samples.

- We can now proceed to create a *ComputingTrait* instance. Precisely, a *ComputingTraitAdjustment* instance.

Now we have everything we need to compute the adjustment, like explained in the section "How to compute" (4.5.1).
When the computation is done, we can browse the response given by the *ComputingComponent* and build the expert system response with the *Response* subclass *AdjustmentResponse*.

Chapter 5

Tests

This chapter aims at presenting the testing tools and procedures that were used in this project.

Unfortunately no tests were done for this project, even though it was planned to do so.

5.1 Tools

5.1.1 Postman

Postman is a software that helps us to test our API. It allows us to forge HTTP requests easily as well as send a collection of requests to the server a multitude of times in a scheduled fashion. This was very helpful to see if the server worked correctly, but it can also be used to make a load test. Unfortunately, no test were done for this project.

Chapter 6

Conclusion

6.1 Planning

6.1.1 Initial planning

This planning was done at the very beginning of the project. Not a lot of analysis was done so it was pretty difficult to judge what tasks will have to be done and in what time proportion.
The main focuses are the implementation of the expected behaviours in the REST API endpoints and the unit tests. The report takes also more time than most of the other tasks.

The initial planning figure can be found in the appendixes (A).

6.1.2 Intermediate planning

The goal of this planning was to show what to expect in the future of the project at the moment of the return of the intermediate report. Most of the analysis and design were already done, but no implementation was done yet.
It was planned to take more than half of the time on the implementation of the expert system features, the related tests and the final report. The rest of the time was planned for the serialization/deserialization of the queries and the responses, the REST API specification, the generation of the server stub and specification of the error and warning messages.

The intermediate planning figure can be found in the appendixes (B).

6.1.3 Final planning

This planning was done for the return of this report to show what was done to compare it with the initial planning.

It is pretty difficult to compare this planning the the initial planning because they are not thought in the same way. However we can still see that the analysis and design part of the project were clearly underestimated. If we go by this planning then the server should have already be implemented by the intermediate return. In reality, not a single implementation was done at this date.

Now if we compare the final planning with the intermediate planning we can see that overall most of the activities took more time than predicted. The one being the most problematic being the implementation of the expert system features. In fact, only one feature is implemented, but it still does not work as to the return of this document. It is close to be finished, but not completely.
A second feature was prepared, but is not as well implemented as the first one.

In the final planning we can where most of the time was taken: the deserialization of the queries, the creation of the REST server, the dosage adjustment feature implementation and the writing of this final report. Some other tasks also took more time than planned, but it is not as critical as the tasks mentioned before.

The final planning figure can be found in the appendixes (C).

6.2 Conclusion

Looking back at what was done, I can see that a lot of time was spent on the analysis and the design of the server. Even more so, most of the time (be it analysis, design or implementation) was spent on the communication between the client and the server. I think I took too much time, thinking over and over about how to make this communication in the most perfect way. Of course the intention is good, but maybe that it could have been done faster. For this, I should maybe have tried with something more simple. That would have allowed me to begin to code sooner.
I noticed that I understood way better the program and how it is supposed to work when I began to code the *Computer* and *AdjustmentComputer* classes. Maybe that if I began to code these classes sooner, I could have had a better overall understanding of the software sooner, which means that I could have updated some of my analysis and design based on the experience I got from coding what the server is actually supposed to do.
However, with the methodology that I used for this project, I almost never had to go back on what I had already done and that is something that I am very happy about. It is difficult to judge if gaining some experience would have made me work faster than the way I did in the end.

I think that one of the things that could have slowed me down is the fact that I am used to work in a team and that this project was done alone. Of course I could always count on the feedback from my supervisor, but I always had the feeling that I hadn't done enough research before asking him or that I didn't try enough. It probably is a bad habit, but I can clearly see now that it is the way I worked in this project.

Another thing I think I did wrong is I was not focused on a single task in particular at the beginning of the project. I noticed that I worked better when I only had one thing to do at a time, but I noticed this only near the end of the project. Being focused on a single task helps me to be more confident in my understanding of the task because took more time working on it. It also keeps me motivated because I understand where I am going with my work and I want to finish it.

I feel like there was a lot of work for this project, but I think that if I could have reduced the time I spent on the communication between the client and the server, especially on the deserialization of the query, then I could at least have finished the dosage adjustment feature. I could also probably have progressed further for the first dosage feature since it is based on the dosage adjustment feature.
I think that maybe if I started to code earlier, then I could have done or at least progressed through more of the expert system feature. However, I think that I probably should have returned to my previous work because it would not have been as polished as it is as I have done it for this project.

Chapter 7

Authenticity

I, Nadir Benallal, hereby declare that the present document is the product of original research and that no other sources than the ones referenced in the bibliography were used.

Date

Signature

Glossary

API Application Programming Interface. 12

ATC Anatomical Therapeutic Chemical is a classification system for active ingredients of drugs. 28

Deserialization is the opposite of serialization. 13

Expert System is an artificial intelligence system based on a set of rules. 6

Getters Method whose goal is to return a member of the instance. 42

GUI Graphical user interface. 5

HL7 Health Level 7 is a specification for communication between hospital's administrative and financial softwares. 12

HTTP Hypertext transfert protocol. 7

IP Internet protocol. 38

IT Information technology. 10

MIC Minimal Inhibitory Concentration. 16

Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. 13

Singleton Oriented Object Programmation design pattern whose objective is to instantiate only one object of a class. 38

Stub is a function (in programming) that has the expected signature, but an incomplete implementation. 12

TDM Therapeutic drug monitoring. 6

UI User Interface. 12

URI Uniform Resource Identifier. 7

XML Extensible Markup Language. 12

XSD XML Schema. 12

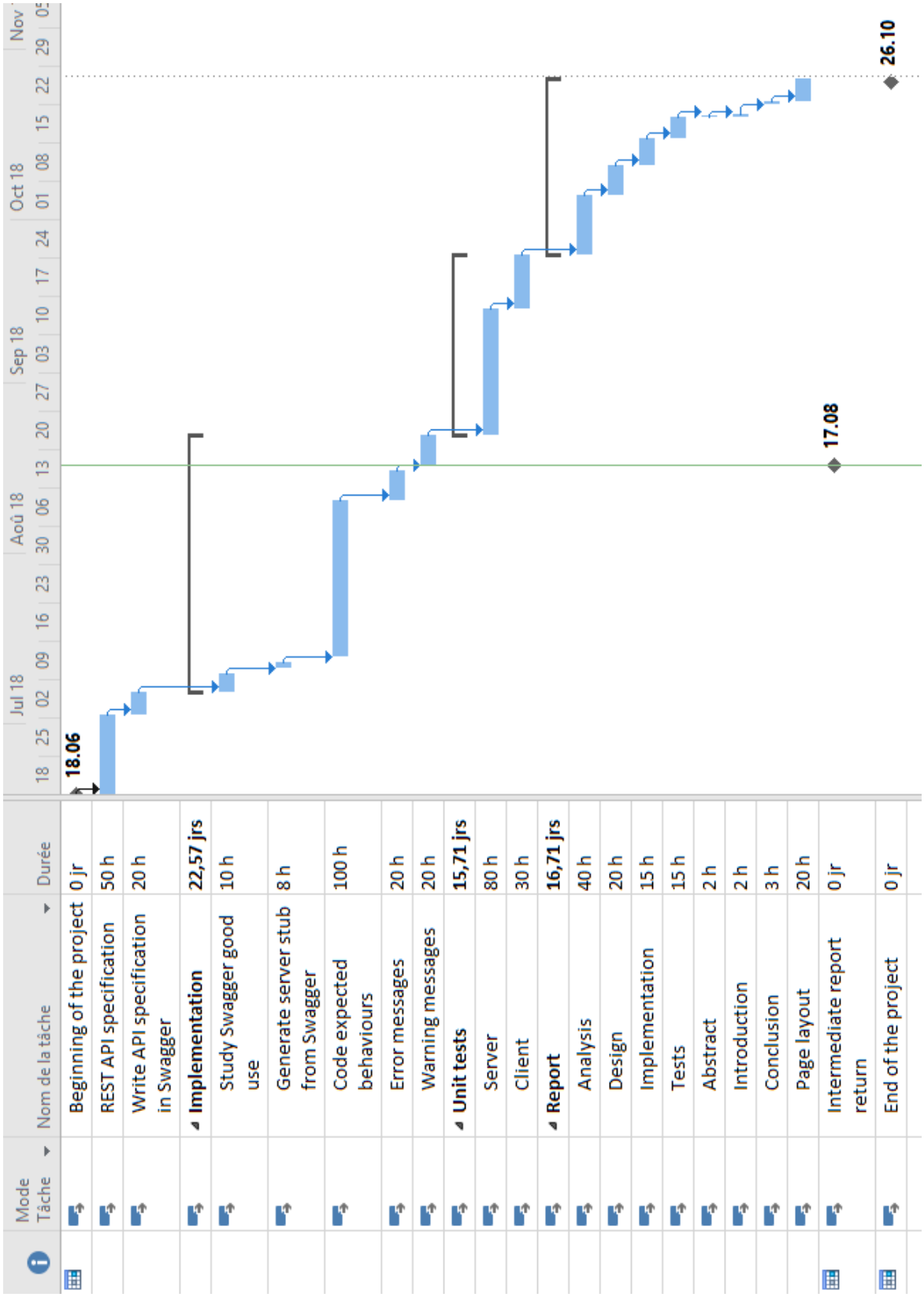
YAML is a human friendly data serialization standard. 12

List of Figures

Appendices

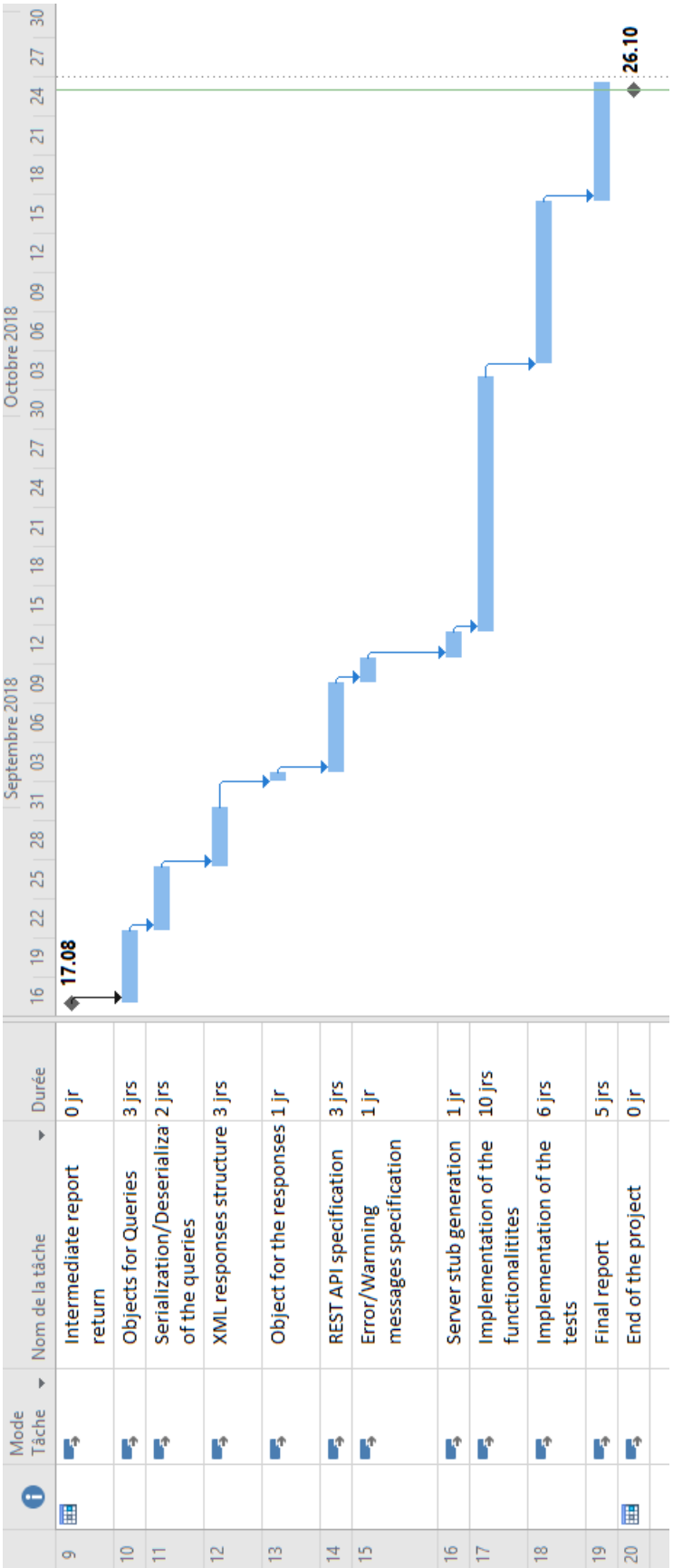
Appendix A

Initial planning figure


Mode Tâche
Nom de la tâche
Durée
Beginning of the project 0 jr
REST API specification 50 h
Write API specification in Swagger 20 h
Implementation 22,57 jrs
Study Swagger good use 10 h
Generate server stub from Swagger 8 h
Code expected behaviours 100 h
Error messages 20 h
Warning messages 20 h
Unit tests 15,71 jrs
Server 80 h
Client 30 h
Report 16,71 jrs
Analysis 40 h
Design 20 h
Implementation 15 h
Tests 15 h
Abstract 2 h
Introduction 2 h
Conclusion 3 h
Page layout 20 h
Intermediate report return 0 jr
End of the project 0 jr
Jul 18
Août 18
Sep 18
Oct 18
Nov
18.06
17.08
26.10

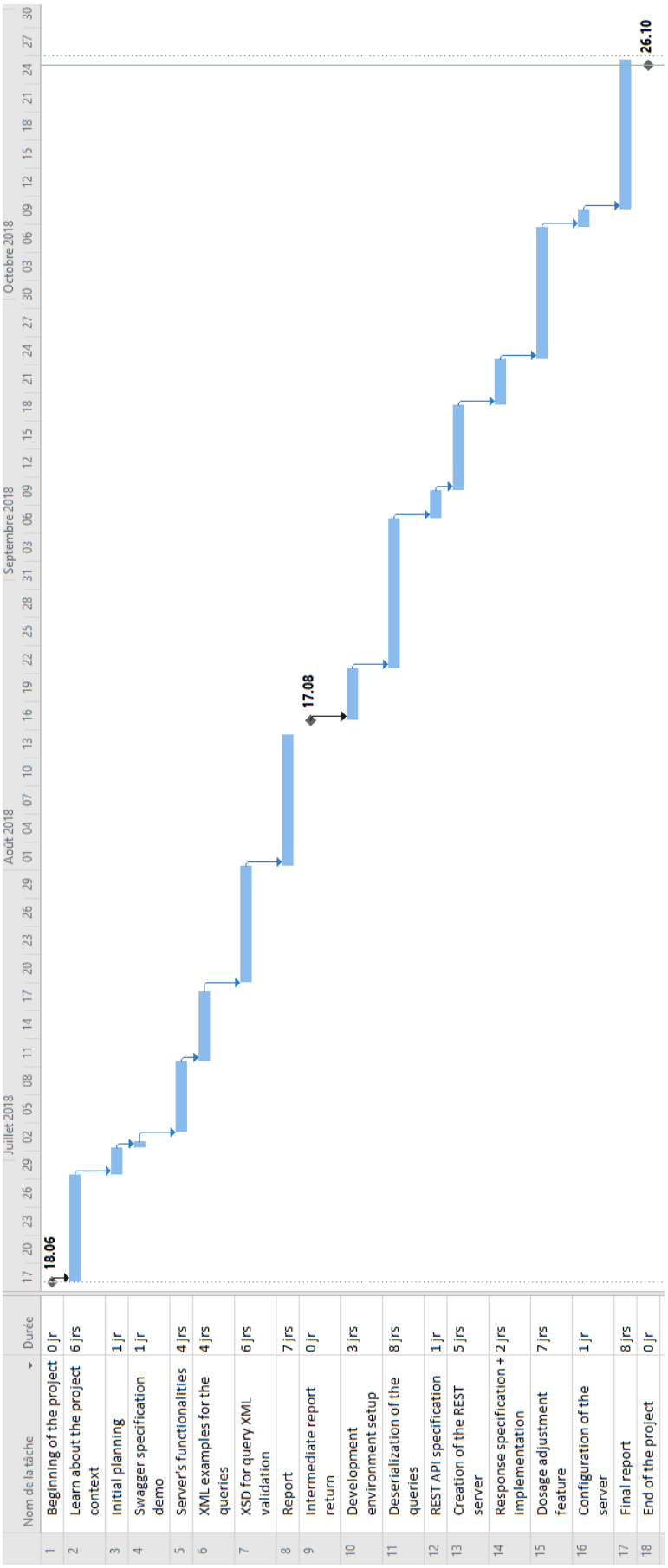
Appendix B

Intermediate planning figure


Mode Tâche
Nom de la tâche
Durée
9 Intermediate report return 0 jr
10 Objects for Queries 3 jrs
11 Serialization/Deserializa of the queries 2 jrs
12 XML responses structure 3 jrs
13 Object for the responses 1 jr
14 REST API specification 3 jrs
15 Error/Warnning messages specification 1 jr
16 Server stub generation 1 jr
17 Implementation of the functionalitites 10 jrs
18 Implementation of the tests 6 jrs
19 Final report 5 jrs
20 End of the project 0 jr
Septembre 2018
Octobre 2018
16 19 22 25 28 31 03 06 09 12 15 18 21 24 27 30 03 06 09 12 15 18 21 24 27 30
17.08
26.10

Appendix C

Final planning figure


Nom de la tâche
Durée
1 Beginning of the project 0 jr
2 Learn about the project context 6 jrs
3 Initial planning 1 jr
4 Swagger specification demo 1 jr
5 Server's functionalities 4 jrs
6 XML examples for the queries 4 jrs
7 XSD for query XML validation 6 jrs
8 Report 7 jrs
9 Intermediate report return 0 jr
10 Development environment setup 3 jrs
11 Deserialization of the queries 8 jrs
12 REST API specification 1 jr
13 Creation of the REST server 5 jrs
14 Response specification + implementation 2 jrs
15 Dosage adjustment feature 7 jrs
16 Configuration of the server 1 jr
17 Final report 8 jrs
18 End of the project 0 jr
Juillet 2018
Août 2018
Septembre 2018
Octobre 2018
18.06
17.08
26.10